UiO : Department of Informatics
University of Oslo

Master thesis: A Linux implementation of MulTFRC

Adrian Jørgensen

Master's Thesis Spring 2014

Master thesis:
A Linux implementation of MulTFRC

Adrian Jørgensen

19th May 2014

Abstract

The use of multimedia streaming on the internet is increasing. With this the need for new protocols is created. The Datagram Congestion Control Protocol (DCCP) can support this better than other existing protocols like Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). To keep the multimedia streaming from causing congestion within the network, the use of congestion control algorithms is required. The implementation of DCCP in the Linux kernel supports two such algorithms. The one concerning this thesis it called TCP-Friendly Rate Control (TFRC). This thesis will implement a new congestion control algorithm called MulTFRC. MulTFRC is made to provide a number of cumulative TFRC flows. This relieves application of complexity concerning the use of multiple TFRC flows to provide prioritization for its services.

This thesis followed the steps of the internet draft "MulTFRC: TFRC with weighted fairness" [8] to implement MulTFRC. Evaluation of the implementation showed that the algorithm works, but that there exists flaws in the Linux kernel implementation of DCCP.

Contents

List of Figures

List of Tables

Part I

Introduction

Motivation

Internet bandwidth has increased substantially in availability in modern time. Along with it, so has the rise of online multimedia streaming services. Data transfer for multimedia streaming services is not the same as sending and delivering data files. A data file must arrive at its receiver in the same size and order as it was sent. A receiver will usually never open or use the file until it has been received in its full size and been put together in the correct order. Therefore delay or loss of packets does not interfere with the impression of the file. The only noticeable difference is that the user will receive it a bit slower. This is obvious as the packets are delayed or the sender have to resend lost packets.

The result of delay on the connection with a multimedia stream is considerably more noticeable. If no mechanism is used to handle this, the multimedia service (e.g. video stream) will have to stall and wait for the packets that has been delayed in order to continue. One commonly used solution is to offer buffering of data. Only using a buffer does however introduce other potential problems concerning dropped packets and congestion. It is preferred that the service does not stall but rather just continues after a packet has been lost. The loss of a packet might just give the service a slight flaw, or it might even unnoticeable. For a video stream this could mean a blurry vision for less then a second. Imagine a person having a phone call. As the call experiences some sort of loss or delay in the network, the person would rather listen to worse quality of the voice from the person on the other end, than having to wait a little while for the packets that was lost to be retransmitted, or those that were delayed to be received, before hearing what the person on the other end said. As for the matter of congestion, it is important that the network connection does not receive more data than it can handle. A congested network will drop packets more often. It is therefore important to have a congestion control to make sure each connection does not contribute to introducing congestion in the network. For a multimedia service this means that the quality can not be higher than what the network can handle in order to provide a stable quality of the service.

File transfers usually increase the sending rate quickly to be able to send data as fast as possible. When a packet is lost the sending speed is usually cut in half by the congestion controller before it starts building back up again. Users of multimedia streams prefer to experience a stable quality of the service, rather than a connection which rapidly change the quality or freezes depending on the flow of packets. A congestion control used for multimedia services therefore needs to create a more stable flow of data rather than pushing the limits of the connection. It must provide an algorithm that increase and decrease the flow of data more slowly than a file transfer congestion control algorithm would do whenever a packet is dropped. The connection will seem more consistent and stable this way. It might take more time for the protocol used for media services to reach max capacity. The sending speed will however not decrease a much each time a packet is lost. When the connection is active for a considerable amount

of time, the throughput for both of these protocols could be more or less equal.

In order to provide a solution to the problems concerning multimedia streaming, these protocols and congestion controls needs to be implemented on a level where they can be used by applications and not having to implemented in the applications themselves. Datagram Congestion Control Protocol (DCCP) [7] is a protocol dedicated for sending data in a manner that multimedia services needs. In order to use a network connection a protocol should be fair to other data flows that uses it. The most used protocol today is Transfer Control Protocol (TCP) [10], which is mostly used to send data files. DCCP already have a congestion control called TCP Friendly Rate Control (TFRC) [6] which provide multimedia streams with the capability to send and receive in a suitable manner. It does however not provide the application to control the importance and priority of the stream. If a system runs two multimedia services, and one of them is twice as important, the application will have to implement functionality to scale the sending rate.

Thesis description

This thesis will focus on an implementation of the congestion control called MulTFRC [8] within DCCP in the Linux kernel. MulTFRC provides the same functionality as TFRC [6], although as the name briefly reveals, it also provides the ability to send a defined number of cumulative TFRC streams. This means it will remove the extra complexity needed on the application level to achieve this and puts it into the transport layer. The implementation of this thesis will be made following the internet draft "MulTFRC: TFRC with weighted fairness" [8]. This draft provides necessary changes that needs to be made in the kernel in order to support MulTFRC. The goal is to make a successful implementation and give an evaluation of it. Any problem that occurs with the implementation or evaluation will be discussed.

Chapter 1

Background and related work

When a network is congested it means that there is too much traffic on the network for it to handle. A congestion control is responsible for making sure that flows using a network does not cause a congestion. When for instance a loss of packets happens in a network, it could mean that the network is congested. The most common way a congestion control adapt to this is by slowing down the rate of which data are sent.

Most traffic on the internet uses some sort of congestion control in order to adapt the flow of data to the restrictions and capacity of the network they are using. Transmission Control Protocol (TCP) does has quite a few implemented congestion controls in the Linux kernel. E.g. Cubic and Reno are two fairly known. The Datagram Congestion Control Protocol (DCCP) is considered experimental within the Linux kernel. Thus it does not have as many implemented congestion controls as TCP. In fact, the Linux kernel does only offer two implemented congestion controls. One of these are the TCP Friendly Rate Control (TFRC).

The next few sections will give a brief explanation of information useful for understanding the content of this thesis and why MulTFRC is useful.

1.1 OSI-Model

The OSI-model consists of 7 layers, shown in figure 1.1, and was created as an international standardization for connections between systems. The bottom layer in the OSI-model is the physical layer. It is responsible for transmitting raw bits from the current system to another. This means it has to decide how much volts should be used to represent 1 and 0 bits, and how to interpret received volts into bits. The second layer to the bottom is the data link layer. Its duties is to create data frames of the input data and deliver it to the above layer, and it also has to make sure it does not drown the buffer on the receiver with data. The network layer handles how packets are routed from source to destination.

The transport layer is of most interest concerning this thesis, as the implementation will reside within it. The function of this layer is to accept data from the above layer and split it up if needed. Further more, it gains responsibilities depending on which protocol is used. I.e. the user can

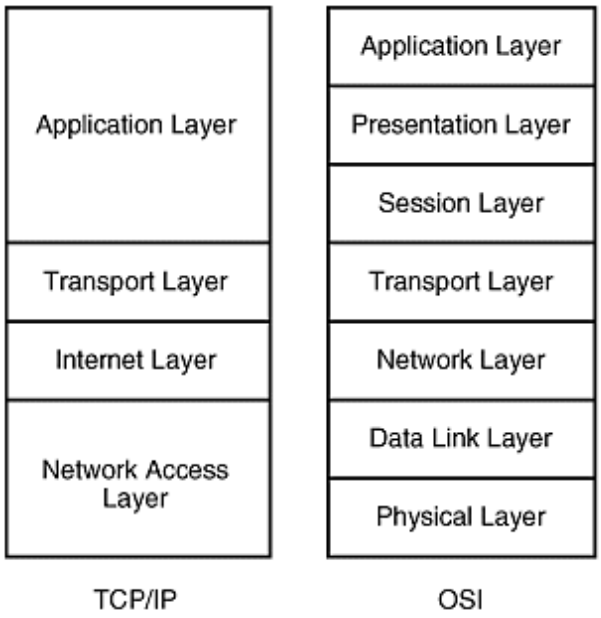


Figure 1.1: TCP/IP and OSI models

decide weather this layer should deliver bytes and packets in the exact same order and at the same time guarantee that the receiver got all of the data, or the user can choose a protocol that has some or none of these properties. Relevant protocols will be explained in section 1.3.

The session layer enables users to create sessions between each other and then keeps track of whose turn it is to transmit, tokens and synchronization. The presentation layer presents the bits and bytes from the lower layers in a way that the above layer requests them. This is done so machines with different representations of data can communicate. And the last layer of the OSI-model is the application layer. Different protocols are used here than in the transport layer. These protocols are used by the users. E.g. HTTP (Hyper Text Transfer Protocol) [3] which is used to download web pages. When the web page is downloaded the transport layer will use TCP to control that every packet is received in the correct order.

Today it is not unusual to skip the session and presentation layer. And as you can see in figure 1.1, they are not represented in the TCP/IP-model. The TCP/IP-model has also combined the data link layer and the physical layer into a host-to-network, and they have renamed network to internet. The focus of the TCP/IP-model lies within the three top layers. E.g. IP (Internet Protocol) on the internet layer, TCP on the transport layer and HTTP on the application layer.

A typical router will use the three bottom layers within the OSI-model or the two bottom ones in the TCP-IP-model, while a typical end system will use all the layers in the TCP-IP-model.

1.2 Linux

In 1991 a Finnish student named Linus Torvalds released the first version of the source code now known as the Linux kernel. His motivation for doing so were based on frustration concerning licensing and also because of his interest in operating systems. Today the Linux kernel is used as a base for many different operating systems. Those systems are called Linux distributions. Debian is a fairly popular distribution and are also the one used to run the kernel implementation of this thesis.

1.2.1 User space

The definition of *User space* is all code that does not run within the kernel. This means that all programs like terminals, web browsers and editors, to mention some, runs in *User space*. Libraries of various kinds exists in *User space*. E.g. the C library, with *open*, *exec* and so on. Advanced graphic drivers and system daemons for sound, and the like, also run in *User space*.

1.2.2 Kernel space

The responsibility of *Kernel space* are to present system calls of the Linux kernel. E.g. file systems and network subsystems. It handles process scheduling and memory management. This thesis will be implemented in the network subsystem of the Linux kernel.

1.3 Protocols

There are two protocols which are mainly used for transporting data in computer networks. The first, Transmission control Protocol (TCP) [10], as the connection oriented and the second, User Datagram Protocol (UDP) [11], as the connectionless. The implementation of this master thesis will not use either of these, but instead the rather new Datagram Congestion Control Protocol (DCCP) [7]. In the next sections a brief introduction will be given of these three.

1.3.1 Transmission Control Protocol

Transmission Control Protocol (TCP) [10] was one of the first available protocols for sending data files between end systems. Today it is the most used protocol on the internet. It provides all the basic abilities an application needs to send data files.

At first a connection is established between the sender and the receiver. The connection set up is done with what is called a 3-way handshake. It is called a 3-way handshake because 3 successfully sent packets are needed to establish the connection. A connection establishment uses two bits in the TCP header, namely the synchronize bit (SYN) and the acknowledge bit (ACK). As an example lets say we have a host A and a host B. Host A first send a packet with the SYN bit set to 1 to host B. Host b accepts the

connection and sends back a synchronize-acknowledgement (SYN-ACK) packet. A SYN-ACK packet means that both SYN and ACK bits are set to 1. When host A receives the SYN-ACK packet it knows host B has accepted the connection and sends back a packet with ACK bit set to 1 to tell host B that the connection is established. When host B receives the ACK packet the connection is considered established at both ends.

When the connection has been established the sender starts to send data to the receiver. The receiver then have to acknowledge that it receives the data. Depending on what method for sending is being used the acknowledgements can be sent back often or more rare. It is usual to acknowledge packets in a certain window. E.g. a sender can send 20 data packets and the receiver then only needs to send an acknowledgement for packet number 20. If it does not receive packet number 20 it will send an acknowledgement for the packet received with the highest sequence number. Because TCP sends acknowledgements of packets received it is a reliable protocol.

The termination of a connection is done with a 4-way handshake, which means 4 successfully sent control packets.

TCP also orders the packets before sending it to the application. If a packet is lost, the transport layer has to make the sender retransmit that packet and wait for it. It is guaranteed that the receiver application will get the data in the exact order it was sent. To combat congestion, the connection between transceivers is altered rapidly to respond to loss or delay. When allowed by the network, TCP will also quickly increase the sending rate for paths with high bandwidth. In short, TCP provides the most basic functionality which is needed for transfer of data files. And this is what makes it popular.

MulTCP

MulTCP is a protocol that offers n cumulative TCP flows and provides this as one interface to the application layer, therefore removing the complexity behind creating more than one flow within the application. The idea itself is great but there has not been any successful attempts to push this as a replacement for the current use of TCP.

1.3.2 User Datagram Protocol

User Datagram Protocol (UDP) [11] are together with TCP the most used transport protocols on the internet. UDP offers a connectionless transport of data for the application layer, thus it gives more control to the application than TCP. A connectionless service does not need any set up or tear down of the connection. The sender application sends a packet and the receiver application can decide on what to do with it. If the packet is some sort of request, the receiver application will most likely respond. Not response is sent from the transport layer.

Every application has to implement their own algorithms and mechanisms to control the flow of data. This does give the potential for less over-

head than TCP would offer, but is also more complex for the application itself. To date, it is mostly used for applications that happens in real time where lost packets are okay. E.g. video call and various types of real time games. It is also useful in some client server situations. For instance where no connection set up is required and the client sends a request that expects a short reply. If the client does not receive a reply it can simply send a new request after a time out. In the case of Domain Name System (DNS) [9], only two packets are sent. (Request and reply) The network overhead in a connection set up and tear down for this kind of communication would increase the amount of packets sent several times over.

The UDP packet consist of source- and destination port, length and checksum. This means it can forward the packet to the application that is attached to the given port and validate the integrity of the packet. The checksum is however optional and in some cases, like digitalized speech where quality does not matter as much, using the checksum field is not necessary.

1.3.3 Datagram Congestion Control Protocol

Datagram Congestion Control Protocol (DCCP) [7] was created as a better alternative than UDP for services that does not need all of what TCP offers. It has reliable set up and tear down of connections like TCP, but uses unreliability for its data flows while providing congestion control. This means it could be a better alternative for multimedia streaming services than both UDP and TCP. It was standardized in 2006 and even so it is rarely used. This might be due to the fact that it is not natively supported in any of the big operating systems. The Linux kernel contains an implementation of DCCP. It is however marked as *experimental*, which means *Prompt for development and/or incomplete code/drivers*.

The congestion control algorithms of DCCP are called CCIDs (Congestion Control IDentifiers) in the Linux kernel. There is only two official CCIDs implemented currently. They are called the TCP-like congestion control as CCID 2 and the TCP-Friendly rate control (TFRC) as CCID 3.

1.3.4 TCP-Friendly Rate Control (TFRC)

Flows have to share the available bandwidth on most networks. Thus it is common to have flows adapt to changes and share the network equally. TCP-Friendly Rate Control (TFRC) [6] is a congestion control algorithm in DCCP that is specifically made to fill the needs of real time communication and the like. It is designed to share the bandwidth of a network fairly with TCP, hence the name. TFRC is also designed to have much less variation of the sending rate than TCP and that is one of the factors that makes it more suitable for real time communication.

While offering a more stable sending rate there is also a downside to TFRC compared to TCP. If changes occur in the network that affects the sending rate, TFRC is much slower at responding to this than TCP. This implies that it is not well suited for sending much data in a short amount of

time, and to use it for anything else than applications that requires a smooth sending rate would be a waste when there are TCP-like alternatives.

TFRC is made to calculate some of the information on the receiver side of a connection. This potentially relieves the server with more CPU and memory to serve more concurrent connections.

TFRC algorithm

To calculate the sending rate TFRC uses the round trip time, packet size and loss event rate. A loss event is defined as a window of data (round trip time) where one of the packets is lost or when Explicit Congestion Notification (ECN) [12] is marked on one of that packets. The last 8 loss events are used to calculate and obtain the smoothness needed for TFRC. The loss event rate is calculated on the receiving end of the connection. The sender uses the round trip time of the packets sent back from this calculation to find the round trip time. The packet size is obviously known to the sender. And as all of the necessary information is available to the sender it can now calculate the sending rate with the congestion control algorithm. To be fair to TCP flows, the TFRC equation must use one of the equations for finding TCP throughput as a function of round trip time and the loss event rate. The throughput equation for Reno TCP is used in the Linux kernel to achieve this.

1.3.5 MulTFRC

As one might interpret from the name, MulTFRC [2] extends the abilities of TFRC by providing the equivalent of multiple TFRC flows. The idea is to offer a protocol that can provide N number of cumulative TFRC flows with only one connection socket, thus providing the bandwidth of N connections with one interface to the application. If an application wants the bandwidth of two TFRC flows, it can simply set up the connection socket with MulTFRC and the value of N being 2, rather than setting up two TFRC connections within the application and having complex merging and splitting mechanisms to combine these. A lot of the complexity concerning connection set up is then removed from the application level and put into the transport layer. This makes it easier to create all applications that requires bandwidth of more or less than one TFRC flow to keep its flow prioritized among other flows. While MulTFRC removes some application complexity it would also remove a mentionable amount of packet overhead and should therefore also be a better solution network wise. E.g. two TFRC flows would have the overhead of 2 flows while MulTFRC with $N = 2$ would have the overhead of one flow but provide the sending rate of two.

Priority for a MulTFRC flow set with N less than one would make it less important than other TCP-like flows. A higher value than one would make it more prioritized than TCP-like flows. And N equal to one would make it equally important.

The internet draft "MulTFRC: TFRC with weighted fairness" [8], written by Dragana Damjanovic and Michael Welzl, describes where MulTFRC is different than TFRC and the necessary changes and steps needed to convert the TFRC implementation of the Linux kernel into MulTFRC.

MulTFRC algorithm

At first glance one might think the result of the TFRC equation could be multiplied by N to achieve the desired effect of MulTFRC. This would however not suffice. The loss event probability would be needed for N flows but there is in fact only one flow. MulTFRC calculates the amount of losses in a loss event. This amount is represented by the variable called j. Together with the rest of the needed variables from the TFRC equation it can be used to calculate the sending rate for N cumulative flows. MulTFRC uses floating-point arithmetic for calculation precision.

The algorithm of MulTFRC has been proven to work in the PhD thesis "Parallel TCP Data Transfers: A Practical Model and its Application" [1] by Dragana Damjanovic.

Part II

The project

Chapter 2

Planning the project

Getting to know where in the kernel the code is suppose to be applied is a natural start. According to RFC 4342 [4] the Linux kernel implementation of TFRC is called CCID 3. While looking through the kernel with tools like *grep* and *vim*, references to TFRC are mostly found in *net/dccp/ccids*. The algorithm for TFRC is located in the library folder for CCIDs, more precisely in the file called *net/dccp/ccids/lib/tfrc_equation.c*. It seemed likely that the equation for MulTFRC should be put in the same place and this is where the work towards a MulTFRC implementation started. To have an easy way to compare both TFRC and MulTFRC for evaluation, a decision to copy all code for TFRC (CCID 3) and create a MulTFRC version that worked equally were the first step. It would be easier to switch congestion control algorithm on a running system than having to reboot into a original kernel between tests. As there are rumoured that someone is already working on a CCID 4 it was decided that the copy will be named CCID 5.

After the copy has been made, changing it to meet the requirements of MulTFRC is the next step. The document "MulTFRC: TFRC with weighted fairness" [8] provides detailed instructions of what has to be changed for MulTFRC to work. These changes are base upon RFCs ([4] and [5]) about the implementation of TFRC in Linux. Using these RFCs is also useful in the matter of locating these fragments of code that needs to be changed, as "MulTFRC: TFRC with weighted fairness" [8] is specifically referring to sections in these RFCs.

2.1 Compilation

The Linux kernel in its entirety needs to be compiled in order to run the implemented code. The first time one does this it takes quite some time to finish the compilation. After the kernel has subdued a full compilation once, new changes to the kernel will only have to be recompiled with certain necessary and affected parts. To relieve the time and effort it takes to reboot a machine in order to test the newly compiled kernel, use of a virtual machine is much valued. A virtual machine also provides some extra features which is nice for working with Linux kernel implementation.

Besides writing to kernel log and other defined prints to files on the host, it also provides an easy set up to check that the implementation works in a virtual network. E.g. two virtual machines is much easier to set up using the custom compiled kernel than two physical machines.

2.1.1 menuconfig

There are several tools one can use to configure the Linux kernel before it is compiled. The maybe most commonly used tool is the *menuconfig*. When all required tools are install on a system *menuconfig* can be started from the root folder of the kernel by running the command below.

$ make menuconfig

This will present a categorical menu where the user can navigate around and tweak the kernel to what ever configuration is required. There are three main choices that can be made for each option. *y* means include feature in kernel, *m* means compile feature as a module loadable by the kernel during runtime and *n* means do not include in the kernel. All the presented choices are configured in the *Kconfig* files of the kernel and the building of the kernel is done by the help of makefiles.

2.1.2 Makefile

Make is a program that reads makefiles and do the instructions they contain. Usually the instructions are aimed at building executable programs. The Linux kernel is built by the instructions of a makefiles located at various directories throughout the kernel. *menuconfig* can be used to configure what parts of the kernel these makefiles build. Actually, the *menuconfig* is also a target of the *Make* utility itself.

Building in Debian

Debian has tools that makes it easy to compile and install the kernel within a common Linux operating system. For that reason it was chosen as a platform for the test machines. Version 6.0.5 of Debian was the most recently released operating system at the time of the implementation. For consistency this version, along with the long-term stable release version 2.6.32 of the kernel, was used during the implementation and evaluation of this thesis. The kernel itself can be fetched from various different sites on the internet. The easiest way to get it using Debian is by running the following command.

$ *apt-get install linux-source-2.6.32*

Other packages needed in order to compile the kernel in Debian:

build-essential, kernel-package, bzip2 and libncurses5-dev

Thereafter a compressed kernel is located at */usr/src/linux-source-2.6.32.tar.bz2*. The next step is to extract it and then configure the Linux kernel source for compilation. This can be done with a tool called *menuconfig* by running the follow command in the root folder of the extracted Linux kernel.

$ *make menuconfig*

From the presented menu the user can enable and disable features of the kernel as pleased. The features for DCCP obviously needs to be enabled to compile the DCCP protocol and the implementation of MulTFRC. The DCCP option can be found under *Networking support* and then under *Networking options*. There the option *The DCCP Protocol (EXPERIMENTAL)* can be set. If it is set with * (Marked) it will be build in the kernel during compilation and if it is set with *M* it will be compiled and linked to the kernel as a loadable module. If nothing is chosen, the kernel will be built without support for DCCP. Next in order is a kernel compilation. A recognizable name of the compiled image can be given, although grub will automatically pick the most recently installed image as default boot option. The command to compile (with *-dccp-multfrc* appended to the name):

$ *make-kpkg –append-to-version=-dccp-multfrc kernel_image – initrd binary*

And lastly the command for installing the compiled image of the Linux kernel.

$ *dpkg -i linux-image-2.6.32-dccp-multfrc_i386.deb*

The installed kernel can now be chosen during start up from the menu presented by grub.

Chapter 3

Implementation

In order to start the implementation of MulTFRC, a new CCID (CCID 5) must be added. Making sure that CCID 5 works exactly as CCID 3 is important before any alteration towards MulTFRC is made. Running tests and comparing the results of them with each other will determine whether the new CCID 5 has been correctly added. Therefore the first round of kernel implementation will only consist of creating CCID 5 without any of the changes or additions needed by MulTFRC. Finding all chunks of code where CCID 3 is used is crucial for creating CCID 5. As a first version CCID 5 will use TFRC where CCID 3 does.

3.1 Creating a new CCID

To start of the creation of a new CCID an enum in *include/linux/dccp.h* is where the first addition was made in the kernel. The idea behind the use of enums in this file is to provide an symbolic identifier for each CCID. These identifiers are often more readable and memorable than what they represent. E.g. *DCCPC_CCID5 = 5* means that whenever the value *DCCPC_CCID5* is set for an integer, the value set for the integer is in fact *5*. This enum field was added on line number 5 in listing 3.1 and is the only alteration of *include/linux/dccp.h*. In this case it is not all that hard to remember as the value is the id number for CCID 5, but it is still a good practice to use these kind of enums even for easy memorable representations. In contrast, a less readable example could be *DCCPF_SEND_LEV_RATE = 192*. An example usage for *DCCPC_CCID5* can be seen in listing 3.8 on line number 56.

Listing 3.1: include/linux/dccp.h

```
1 /* DCCP CCIDS */
2 enum {
3         DCCPC_CCID2 = 2,
4         DCCPC_CCID3 = 3,
5         DCCPC_CCID5 = 5,
6 };
```

3.1.1 CCID interface

To set up an interface for a specific CCID a struct of *ccid_operations* needs to be declared. If the kernel is compiled with *CONFIG_IP_DCCP_CCID5*, which means it is compiled with CCID 5 enabled, then *net/dccp/ccid.h* needs to know how to call specific functions for CCID 5. An extern struct with CCID 5 specific information will therefore be available after compilation of the kernel in *net/dccp/ccid.h*. (See listing **??**) Thus it is also available for *net/ddcp/ccid.c*. *net/dccp/ccid.c* will activate the CCID given it has been defined and therefore is amongst the available extern structs.

Listing 3.2: net/dccp/ccid.h

```
1 #ifdef CONFIG_IP_DCCP_CCID5
2 extern struct ccid_operations ccid5_ops;
3 #endif
```

Kprobe

The file *net/dccp/probe.c* is used to enable observation of DCCP flows with a program called *kprobe*. To keep things as similar to CCID 3 as possible, a handler for CCID 5 was also added. In listing 3.3 the function that is called whenever a packet is sent with DCCP is shown. A test to check if the CCID version used by the sent packet is CCID 5, has to be added to make this work with CCID 5. (Line number 11 in listing 3.3) When CCID 5 is used a struct (Line number 6 in listing 3.3) with socket information for the sender half will be set with current socket information. Following is a test to check whether this struct is set or not. When it is set, *kprobe* can now non-disruptively check every variable that is printed from this handler. Before MulTFRC is implemented this print is the same as the one for CCID 3, but more information will be sent eventually. Thus more information needs to be printed as well.

Listing 3.3: net/dccp/probe.c

```
1  static int jdccp_sendmsg(struct kiocb *iocb, struct sock *sk,
2                           struct msghdr *msg, size_t size)
3  {
4          const struct inet_sock *inet = inet_sk(sk);
5          struct ccid3_hc_tx_sock *ccid3_hctx = NULL;
6          struct ccid5_hc_tx_sock *ccid5_hctx = NULL;
7
8          if (ccid_get_current_tx_ccid(dccp_sk(sk)) == DCCPC_CCID3)
9                  ccid3_hctx = ccid3_hc_tx_sk(sk);
10
11         if (ccid_get_current_tx_ccid(dccp_sk(sk)) == DCCPC_CCID5)
12                 ccid5_hctx = ccid5_hc_tx_sk(sk);
13
14         if (port == 0 || ntohs(inet->dport) == port ||
15             ntohs(inet->sport) == port) {
16                 if (ccid3_hctx)
17                         printl("%pI4:%u %pI4:%u %d %d %d %d %u "
```

```
18                                      "%llu %llu %d\n",
19                                      &inet->saddr, ntohs(inet->sport),
20                                      &inet->daddr, ntohs(inet->dport), size,
21                                      ccid3_hctx->ccid3hctx_s, hctx->ccid3hctx_rtt,
22                                      ccid3_hctx->ccid3hctx_p, hctx->ccid3hctx_x_calc,
23                                      ccid3_hctx->ccid3hctx_x_recv >> 6,
24                                      ccid3_hctx->ccid3hctx_x >> 6, hctx->ccid3hctx_t_ipi);
25                      else if (ccid5_hctx)
26                              printl("%pI4:%u %pI4:%u %d %d %d %d %u "
27                                      "%llu %llu %d\n",
28                                      &inet->saddr, ntohs(inet->sport),
29                                      &inet->daddr, ntohs(inet->dport), size,
30                                      ccid5_hctx->ccid5hctx_s, hctx->ccid5hctx_rtt,
31                                      ccid5_hctx->ccid5hctx_p, hctx->ccid5hctx_x_calc,
32                                      ccid5_hctx->ccid5hctx_x_recv >> 6,
33                                      ccid5_hctx->ccid5hctx_x >> 6, hctx->ccid5hctx_t_ipi);
34                      else
35                              printl("%pI4:%u %pI4:%u %d\n",
36                                      &inet->saddr, ntohs(inet->sport),
37                                      &inet->daddr, ntohs(inet->dport), size);
38              }
39
40              jprobe_return();
41              return 0;
42      }
```

3.1.2 CCID 5

The next step towards creating CCID 5 is to create both *net/dccp/ccids/ccid5.c* and *net/dccp/ccids/ccid5.h*. As they will not be using MulTFRC, the most important change concerning these to files is in the header file (*net/dccp/ccids/ccid5.h*). This is where the socket options for both the sender half and the receiver half of the connection is defined in structs. See listing **??**. In this version they both have references to TFRC structs. The sender side includes *struct tfrc_tx_info* which contains important information concerning sending rate (*ccid5hctx_x, ccid5hctx_x_recv and ccid5hctx_x_calc*), round trip time (*ccid5hctx_rtt*) and current loss event rate (*ccid5hctx_p*). Packet size (*ccid5hctx_s*) is also contained, although this is set outside of the TFRC struct. The parameters used to calculate the sending rate (*ccid5hctx_x_calc*) are packet size (*ccid5hctx_s*), round trip time (*ccid5hctx_rtt*) and current loss event rate *ccid5hctx_p*. Together with these three parameters N and j will also be needed to calculate the sending rate (*ccid5hctx_x_calc*) with MulTFRC.

Listing 3.4: net/dccp/ccids/ccid5.h

```
1  struct ccid5_hc_tx_sock {
2          struct tfrc_tx_info ccid5hctx_tfrc;
3  #define ccid5hctx_x ccid5hctx_tfrc.tfrctx_x
4  #define ccid5hctx_x_recv ccid5hctx_tfrc.tfrctx_x_recv
5  #define ccid5hctx_x_calc ccid5hctx_tfrc.tfrctx_x_calc
6  #define ccid5hctx_rtt ccid5hctx_tfrc.tfrctx_rtt
```

```
7  #define ccid5hctx_p ccid5hctx_tfrc.tfrctx_p
8  #define ccid5hctx_t_rto ccid5hctx_tfrc.tfrctx_rto
9  #define ccid5hctx_t_ipi ccid5hctx_tfrc.tfrctx_ipi
10         u16 ccid5hctx_s;
11         enum ccid5_hc_tx_states ccid5hctx_state:8;
12         u8 ccid5hctx_last_win_count;
13         ktime_t ccid5hctx_t_last_win_count;
14         struct timer_list ccid5hctx_no_feedback_timer;
15         ktime_t ccid5hctx_t_ld;
16         ktime_t ccid5hctx_t_nom;
17         u32 ccid5hctx_delta;
18         struct tfrc_tx_hist_entry *ccid5hctx_hist;
19         struct ccid5_options_received ccid5hctx_options_received;
20 };
21 struct ccid5_hc_rx_sock {
22         u8 ccid5hcrx_last_counter:4;
23         enum ccid5_hc_rx_states ccid5hcrx_state:8;
24         u32 ccid5hcrx_bytes_recv;
25         u32 ccid5hcrx_x_recv;
26         u32 ccid5hcrx_rtt;
27         ktime_t ccid5hcrx_tstamp_last_feedback;
28         struct tfrc_rx_hist ccid5hcrx_hist;
29         struct tfrc_loss_hist ccid5hcrx_li_hist;
30         u16 ccid5hcrx_s;
31 #define ccid5hcrx_pinv ccid5hcrx_li_hist.i_mean
32 };
```

3.1.3 Kconfig

The file *net/dccp/ccids/Kconfig* needs to add the option of CCID 5 in order for *make menuconfig* to present it as an option and for the *Makefile* to compile it. The choices made with *make menuconfig* is reflected into *net/dccp/Makefile* (Listing 3.6). Basically what the addition in listing 3.5 does is to provide CCID 5 as an option if DCCP is either enabled within the kernel (*IP_DCCP = y*) or as a module (*IP_DCCP = y*). It also provides the option for enabling debugging messages and the option to set a different *nofeedback timer*. The TFRC library needs to be added if kernel is compiled with either or both CCID 3 and CCID 5. On line 29 and 32 in listing 3.5 a test to see whether CCID 3 or CCID 5 is enabled and if so it also enables TFRC library and debug messages. It is possible to give them all a default value, but as this has not been added for CCID 3 it will not be so for CCID 5 either.

Listing 3.5: net/dccp/ccids/Kconfig

```
1  config IP_DCCP_CCID5
2          bool "CCID-5 (MulTFRC) (EXPERIMENTAL)"
3          def_bool y if (IP_DCCP = y || IP_DCCP = m)
4          ---help---
5            CCID-5 denotes MulTFRC, an equation-based rate-controlled
6            congestion control mechanism.
7
8            If in doubt, say N.
```

```
9
10 config IP_DCCP_CCID5_DEBUG
11         bool "CCID-5 debugging messages"
12         depends on IP_DCCP_CCID5
13         ---help---
14           Enable CCID-5 specific debugging messages.
15
16           The debugging output can additionally be toggled by setting the
17           ccid5_debug parameter to 0 or 1.
18
19           If in doubt, say N.
20
21 config IP_DCCP_CCID5_RTO
22           int "Use higher bound for nofeedback timer"
23           default 100
24           depends on IP_DCCP_CCID5 && EXPERIMENTAL
25           ---help---
26             Use higher lower bound for nofeedback timer expiration.
27
28 config IP_DCCP_TFRC_LIB
29         def_bool y if (IP_DCCP_CCID3 || IP_DCCP_CCID5)
30
31 config IP_DCCP_TFRC_DEBUG
32         def_bool y if (IP_DCCP_CCID3_DEBUG || IP_DCCP_CCID5_DEBUG)
```

3.1.4 Makefile changes

During compilation of the kernel the *net/dccp/Makefile* is called to build all necessary files and optional files needed by the enabled options chosen with *menuconfig*. All options with *dccp-y* will be built. With some exceptions, most of the options in *net/dccp/Makefile* is determined by *menuconfig*. CCID 2 is enabled by default whenever DCCP is compiled and has *dccp-y* hard coded. CCID 5 will be an optional choice in *menuconfig* and the option in *net/dccp/Makefile* is defined by *dccp-$(CONFIG_IP_DCCP_CCID3)* as either *y* (built in kernel) or *m* (built as module). Any files added for CCID 5 also have to be added in *net/dccp/Makefile*. Before MulTFRC is added only line number 8 in listing 3.6 has to be added.

Listing 3.6: net/dccp/Makefile

```
1 dccp-y += ccids/ccid2.o ackvec.o
2 dccp-$(CONFIG_IP_DCCP_CCID3) += ccids/ccid3.o
3 dccp-$(CONFIG_IP_DCCP_TFRC_LIB) += ccids/lib/tfrc.o \
4                                    ccids/lib/tfrc_equation.o \
5                                    ccids/lib/packet_history.o \
6                                    ccids/lib/loss_interval.o
7
8 dccp-$(CONFIG_IP_DCCP_CCID5) += ccids/ccid5.o
```

3.1.5 Feature negotiation

The file *net/dccp/feat.c* is responsible for feature negotiation concerning the set up of a DCCP connection. It is specified in RFC4340 [7], Section 6. A few lines of code has to be added in this file to provide a feature negotiation for CCID 5.

Listing 3.7: net/dccp/feat.c

```
1 int sysctl_dccp_rx_ccid __read_mostly = 5;
2 int sysctl_dccp_tx_ccid __read_mostly = 5;
```

To make things easier during testing CCID 5 were set to default CCID for DCCP. This feature is achieved by setting the two integers in listing 3.7 to the value of 5. This ensures that CCID 5 is the default congestion control mechanism for both sending (*sysctl_dccp_tx_ccid*) and receiving (*sysctl_dccp_rx_ccid*) packets when the DCCP protocol is used. These kind of variables are the default value of kernel parameters and can be found in various places within the kernel. They can also be changed during runtime. The previously default value were 3 (CCID 3). Whenever CCID 3 needs to be tested it can simply be set by the use of two commands, one for sending and one for receiving. Arguably this could be left untouched, although it is more than likely that CCID 5 will be the most used CCID during this implementation and testing of this thesis. Therefore this was done to avoid setting the sysctl variables after each boot in order to run tests with CCID 5. I.e. *sysctl_dccp_rx_ccid* and *sysctl_dccp_tx_ccid* has to be set to 3 in order to run tests with TFRC (CCID 3) to compare them with MulTFRC (CCID 5). To set these variables with the terminal either the kernel has to be compiled with DCCP built in or the DCCP kernel module has to have been used or switched on with *modprobe*. After this has been done *sysctl* must be called with the write parameter *-w* to set a new value for the kernel parameter. An example of how one can change the kernel parameters to use a different congestion control algorithm in DCCP than what is currently used is shown below.

$ modprobe dccp (Needed if DCCP is compiled as a module)

$ sysctl -w net.dccp.default.rx_ccid=3

$ sysctl -w net.dccp.default.tx_ccid=3

The next change in *net/dccp/feat.c* are the dependencies that describes properties for the connection on each side of a connection. E.g. *DCCPF_SEND_ACK_VECTOR*. These properties should be the same in CCID 5 as in CCID 3. It is clearly stated in the comments that each CCID should have its own corresponding dependency table, and therefore a table equal to CCID 3 was made instead of returning the same table for CCID 5. See code in listing 3.8.

Listing 3.8: net/dccp/feat.c

```
1 static const struct ccid_dependency ccid5_dependencies[2][5] = {
```

```
2           { /* Dependencies same as CCID3 */
3               {
4                   .dependent_feat = DCCPF_SEND_ACK_VECTOR,
5                   .is_local = true,
6                   .is_mandatory = false,
7                   .val = 0
8               },
9               {
10                  .dependent_feat = DCCPF_SEND_LEV_RATE,
11                  .is_local = true,
12                  .is_mandatory = true,
13                  .val = 1
14              },
15              {
16                  .dependent_feat = DCCPF_SEND_NDP_COUNT,
17                  .is_local = false,
18                  .is_mandatory = true,
19                  .val = 1
20              },
21              { 0, 0, 0, 0 },
22          },
23          {
24              {
25                  .dependent_feat = DCCPF_SEND_ACK_VECTOR,
26                  .is_local = false,
27                  .is_mandatory = false,
28                  .val = 0
29              },
30              {
31                  .dependent_feat = DCCPF_SEND_LEV_RATE,
32                  .is_local = true,
33                  .is_mandatory = true,
34                  .val = 1
35              },
36              {
37                  .dependent_feat = DCCPF_ACK_RATIO,
38                  .is_local = true,
39                  .is_mandatory = false,
40                  .val = 0
41              },
42              {
43                  .dependent_feat = DCCPF_SEND_NDP_COUNT,
44                  .is_local = true,
45                  .is_mandatory = false,
46                  .val = 1
47              },
48              { 0, 0, 0, 0 }
49          }
50      };
51      switch (ccid) {
52      case DCCPC_CCID2:
53          return ccid2_dependencies[is_local];
54      case DCCPC_CCID3:
55          return ccid3_dependencies[is_local];
```

```
56      case DCCPC_CCID5:
57          return ccid5_dependencies[is_local];
58      default:
59          return NULL;
60      }
```

The dependencies for the receiver side is the first block (Line 3 to 22 in listing 3.8) and the dependencies for the sender side follows in the next block (Line 23 to 50 in listing 3.8). The dependency *DCCPF_SEND_ACK_VECTOR* for the receiver has *.is_local* set to true and *.val* set to *0*. This means that sending of acknowledgement vectors is disabled locally. However, the sender side has *.is_local* set to false. This means that the sender side does not disable reception of acknowledgement vectors, but if they will be ignored within the underlying congestion control algorithm. On the sender side *DCCPF_ACK_RATIO* is also set with a value of *0*. Both of these dependencies are naturally disabled as acknowledgement vectors are not used in CCID 3 or CCID 5.

The dependency for sending and receiving loss event rate is defined with *DCCPF_SEND_LEV_RATE*. For both sender and receiver it is enabled and mandatory. This means they both send it and expects to receive it. It is needed on both sides to determine the sending rate.

The last dependency field is Non-Data Packets (NDP) count, represented by *DCCPF_SEND_NDP_COUNT*. It is used to calculate whether or not data packets was lost during a packet loss situation. This is done by checking the NDP count versus the sequence number of the current packet and the last previously received before the loss happened. This dependency is only sent by the sender side and only expected by the receiver side. Further information about this can be found in section 7.7 of RFC 4340 [7] or in section 6.1 of RFC 4342 [4].

CCID 5 replicated

CCID 5 is now added to the kernel and works in the exact same way as CCID 3. The next step towards a complete MulTFRC implementation is to create library files for MulTFRC and replace all previous references to the TFRC library with them. Any new files also needs to be added in *Makefiles* to make sure they are included during compilation.

3.1.6 Library replication

The first library file to be replicated are *net/dccp/ccids/libs/tfrc.h*. A the obvious name for the new file is *net/dccp/ccids/libs/multfrc.c*. This file consists mostly of linked initiation functions for both sender and receiver parts of a connection. It also contains a linked function to the TFRC equation itself, which can be found in *net/dccp/ccids/libs/multfrc_equation.c*. In listing 3.9 these linked functions can be seen. The TFRC function call with input parameters can be found on line number 1. This linked function has to include two more parameters with the MulTFRC implementation. For now all function links containing **tfrc** will be renamed **multfrc**.

Listing 3.10 shows parts of listing *net/dccp/ccids/lib/tfrc.h* before file and linked functions renaming.

Listing 3.9: net/dccp/ccids/lib/tfrc.h

```
1 extern u32 tfrc_calc_x(u16 s, u32 R, u32 p);
2 extern u32 tfrc_calc_x_reverse_lookup(u32 fvalue);
3
4 extern int tfrc_tx_packet_history_init(void);
5 extern void tfrc_tx_packet_history_exit(void);
6 extern int tfrc_rx_packet_history_init(void);
7 extern void tfrc_rx_packet_history_exit(void);
```

Listing 3.10 shows parts of *net/dccp/ccids/lib/multfrc.h* after renaming.

Listing 3.10: net/dccp/ccids/lib/multfrc.h

```
1 extern u32 multfrc_calc_x(u16 s, u32 R, u32 p);
2 extern u32 multfrc_calc_x_reverse_lookup(u32 fvalue);
3
4 extern int multfrc_tx_packet_history_init(void);
5 extern void multfrc_tx_packet_history_exit(void);
6 extern int multfrc_rx_packet_history_init(void);
7 extern void multfrc_rx_packet_history_exit(void);
```

The same form of renaming for functions, data structures and variables were done to several more files to create a renamed working copy of the TFRC library.

net/dccp/ccids/lib/multfrc.h

net/dccp/ccids/lib/multfrc.c

net/dccp/ccids/lib/multfrc_equation.h

net/dccp/ccids/lib/multfrc_equation.c

net/dccp/ccids/lib/mulpacket_history.h

net/dccp/ccids/lib/mulpacket_history.c

net/dccp/ccids/lib/mulloss_interval.h

net/dccp/ccids/lib/mulloss_interval.c

Adding CCID 5 and MulTFRC to compilation routine

These new library files needs to be added to the compilation routine. A new configuration test for MulTFRC needs to be added to *Kconfig*. The test simply enables the *IP_DCCP_MULTFRC_LIB* and *IP_DCCP_MULTFRC_DEBUG* if CCID 5 (*IP_DCCP_CCID5*) or CCID 5 debug (*IP_DCCP_CCID5_DEBUG*) has been enabled with *menuconfig*. See listing 3.11.

Listing 3.11: net/dccp/ccids/Kconfig

```
1 config IP_DCCP_MULTFRC_LIB
2         def_bool y if IP_DCCP_CCID5
3
4 config IP_DCCP_MULTFRC_DEBUG
5         def_bool y if IP_DCCP_CCID5_DEBUG
```

When the *Kconfig* has been added, the *net/dccp/Makefile* must also reflect this in order to compile these files along with the rest of the kernel. On line 1 and 2 in listing 3.12 the check for the configured *Kconfig* is added. If DCCP and CCID 5 is enabled for the compilation, all of the files in listing 3.12 will be compiled with the rest of the kernel.

Listing 3.12: net/dccp/Makefile

```
1 dccp-$(CONFIG_IP_DCCP_CCID5)  += ccids/ccid5.o
2 dccp-$(CONFIG_IP_DCCP_MULTFRC_LIB) += ccids/lib/multfrc.o   \
3                                       ccids/lib/multfrc_equation.o  \
4                                       ccids/lib/mulpacket_history.o \
5                                       ccids/lib/mulloss_interval.o
```

Data structure conversion

A data structure file for MulTFRC also has to be added. The *include/linux/tfrc.h* will be copied and named *include/linux/multfrc.h*. This header file also only need to change data structure and variable names to suit the MulTFRC standard set in the library files.

After all of the library and data structure files has been converted to a MulTFRC copy of TFRC, without any of the MulTFRC functionality, the CCID 5 files also needs to use these renamed functions. With the use of the converted named functions the kernel now has a new CCID (CCID 5) and is ready to receive the changes needed to implement MulTFRC.

3.2 MulTFRC implementation

Transformation from TFRC to MulTFRC, following the specifications of "MulTFRC: TFRC with weighted fairness" [2], can be applied onto the already implemented replica of CCID 3. "MulTFRC: TFRC with weighted fairness" [2] provides sections of specifications that should be applied to the current implementation of TFRC in the Linux kernel in order to convert it into MulTFRC. The Linux kernel implementation of TFRC has been done following "TCP Friendly Rate Control (TFRC): Protocol Specification" (RFC 5348) [5] and necessary changes are pointed out by their representation of section in this document. Changes that are not specified also has to be implemented as a consequence to this transformation.

3.2.1 Changes to section 3 of RFC 5348

This section of changes contains the most important part of the transformation to MulTFRC. More precisely, the changes to the algorithm that is

needed to calculate the sending rate of MulTFRC. In addition to the input parameters already existing for TFRC this change also requires N (the number of cumulative flows) and j (the number of packets lost in a loss event). The algorithm specified for TFRC can bee seen in listing 3.13.

Listing 3.13: TFRC algorithm

```
1                                    s
2   X_calc = ------------------------------------------------
3            R * sqrt(2*b*p/3) + (3 * t_RTO * sqrt(3*b*p/8) * (p + 32*p^3))
```

The already existing Linux kernel implementation of TFRC has made assumptions about input parameters and made changes to the algorithm accordingly. With a few iterations of breaking down this algorithm it has been transformed into a lookup table and a few mathematical operations. With the use of the MulTFRC algorithm this lookup table is not needed for the calculation of *x_calc*. It is however used to determine the value of *p*, thus it can not be removed. The mechanism for finding *p* can be seen in listing 3.14. The lookup table is shortened as it consist of 500 rows. The *tfrc_calc_x_reverse_lookup* uses the value of *fvalue* to find the closest corresponding value of *p* from the lookup table. *fvalue* is the value of a scaled division of *s* (packet size), *rtt* (round trip time) and *x_recv* (received sending rate).

Listing 3.14: Reverse lookup to find p

```
1  /** TFRC_CALC_X_ARRSIZE equals the value 500, thus 500 rows in lookup table */
2  static const u32 tfrc_calc_x_lookup[TFRC_CALC_X_ARRSIZE][2] = {
3          { 37172, 8172 },
4          { 53499, 11567 },
5          { 66664, 14180 },
6          { 78298, 16388 },
7          { 89021, 18339 },
8          { 99147, 20108 },
9  ...
10 };
11
12 /**
13  * tfrc_calc_x_reverse_lookup - try to find p given f(p)
14  * @fvalue: function value to match, scaled by 1000000
15  * Returns closest match for p, also scaled by 1000000
16  */
17 u32 tfrc_calc_x_reverse_lookup(u32 fvalue)
18 {
19         int index;
20
21         if (fvalue == 0) /* f(p) = 0 whenever p = 0 */
22                 return 0;
23
24         /* Error cases. */
25         if (fvalue < tfrc_calc_x_lookup[0][1]) {
26                 DCCP_WARN("fvalue %u smaller than resolution\n", fvalue);
27                 return TFRC_SMALLEST_P;
28         }
```

```
29          if (fvalue > tfrc_calc_x_lookup[TFRC_CALC_X_ARRSIZE - 1][0]) {
30                  DCCP_WARN("fvalue %u exceeds bounds!\n", fvalue);
31                  return 1000000;
32          }
33
34          if (fvalue <= tfrc_calc_x_lookup[TFRC_CALC_X_ARRSIZE - 1][1]) {
35                  index = tfrc_binsearch(fvalue, 1);
36                  return (index + 1) * TFRC_CALC_X_SPLIT / TFRC_CALC_X_ARRSIZE;
37          }
38
39          /* else ... it must be in the coarse-grained column */
40          index = tfrc_binsearch(fvalue, 0);
41          return (index + 1) * 1000000 / TFRC_CALC_X_ARRSIZE;
42  }
43
44
45  /* return largest index i such that fval <= lookup[i][small] */
46  static inline u32 tfrc_binsearch(u32 fval, u8 small)
47  {
48          u32 try, low = 0, high = TFRC_CALC_X_ARRSIZE - 1;
49
50          while (low < high) {
51                  try = (low + high) / 2;
52                  if (fval <= tfrc_calc_x_lookup[try][small])
53                          high = try;
54                  else
55                          low = try + 1;
56          }
57          return high;
58  }
```

Algorithm

The algorithm used to replace the one of TFRC is recited in listing 3.15. At first sight it seems like a easy substitute, with only the addition of N and j. Together with N and j, s (packet size), R (round trip time) and p (loss event rate) can be used to obtain the rest of the needed variables. It is assumed for MulTFRC, as it is for TFRC, that $b = 1$ and $t\_RTO = R * 4$. The kernel does provide functions to do *max*, *min* and *ceil*. The kernel does also provide means to calculate all the operands but power of and square root. These two calculations needs to be added as the kernel provides no native support for either. This poses a potential problem with the implementation. In the cases where the operation just used *power of 2* ($x\hat{2}$), it is simple enough to swap it with $x * x$. But then there is still one power of operation where j is the exponent. As j can be many different values, a function to support power of has to be made along with a function for square root.

Listing 3.15: MulTFRC initial algorithm

```
1  If (N < 12) {
2      af = N * (1-(1-1/N)^j);
3  }
```

```
4  Else {
5      af = j;
6  }
7  af = max(min(af,ceil(N)),1);
8  a = p*b*af*(24*N^2+p*b*af*(N-2*af)^2);
9  x = (af*p*b*(2*af-N)+sqrt(a))/(6*N^2*p);
10 z = t_RTO*(1+32*p^2)/(1-p);
11 q = min(2*j*b*z/(R*(1+3*N/j)*x^2), N*z/(x*R), N);
12 X_calc = ((1-q/N)/(p*x*R)+q/(z*(1-p)))*s;
```

The Linux kernel does avoid using floating-point arithmetic whenever it is possible, thus integer arithmetic is favoured. This poses a third problem concerning the MulTFRC algorithm. The precision is important to provide an accurate calculation of the sending rate. Unfortunately this precision is acquired by the use of floating-point arithmetic. There are several ways to go at this problem.

Floating-point arithmetic

The use of a Floating Point Unit (FPU) is required to do floating-point arithmetic operations in the kernel. If the system does have a FPU available and the kernel uses it, it might become corrupted for a task running in user space. When the kernel switches context between tasks running in user space, and if used by the task, the FPU is saved along with the rest of the context. This is however not done automatically with system calls within the kernel. *kernel_fpu_begin* and *kernel_fpu_end* should be used to prevent the FPU from getting corrupted. *kernel_fpu_begin* must be called before any floating-point operation and *kernel_fpu_end* must be called after. *kernel_fpu_begin* and *kernel_fpu_end* are both calls implemented in the *asm/i387.h* header file in the kernel.

To use the operations requiring the FPU within the kernel is however not preferred. The main reason is that the kernel should not contain any floating-point arithmetic because not all Central Processing Units (CPUs) supports the use of a FPU. It is therefore a common opinion that any function requiring floating-point arithmetic does not belong in the kernel. Nevertheless, if it could solve the problem it is worth testing. A simple function were made to test and see how these kernel calls work. It does a simple multiplication of *1.5 * 1.5 = 2.25* but if it is interpreted as integers it would be multiplied as *1 * 1 = 1*. The kernel does not allow *printk* to output floats. A conversion to integer has to be made. The right result will therefore be output as *2* instead of *2.25* while *1* would still mean it calculated it with integer values. See listing 3.16.

Listing 3.16: Floating-point arithmetic test function

```
1  #include <asm/i387.h>
2
3  void testFloatingPoint() {
4      kernel_fpu_begin();
5
6      int i;
```

```
 7      float x = 1.5 * 1.5;
 8      i = x;
 9      printk("%d\n", i);
10
11      kernel_fpu_end();
12  }
```

The use of *kernel_fpu_begin* and *kernel_fpu_end* did not turn out to be a success. Whenever the kernel calls the code running this simple test function, it goes into a state of kernel panic causing the machine to stall. The message output of the created stack trace reads:

> *Some program might be trying to access hardware directly.*

This error message means the kernel is trying to access the FPU to do floating-point arithmetic, but a FPU cannot found on the system running the kernel. One can not guarantee the existence of an FPU and therefore the recommendation not to use floating-point arithmetic operations within the kernel is quite clear. If the kernel were to stall every time the MulTFRC implementation is used on a system that does not have FPU capabilities, it is not considered a valid option to use floating-point arithmetic for the algorithm.

A possibility is to make an option in *menuconfig* where the enabling of floating-point arithmetic is allowed. An implementation of a integer version still has to be made, and set as default. But if the system where the kernel is compiled has a FPU, one could enable the floating-point version. This might still not be a solution that works in environments where the same compilation within an operating system is transferred to new virtual machines, considering that the virtual machines might run on different hardware. Therefore some machines might have an floating-point capabilities and some might not.

A second approach, though close to the one mentioned above, could be to enable different parts of the implementation controlled with a kernel configuration parameter (*sysctl*). This would however make the option more exposed to the system and might not be as feasible as the compiled version. The kernel configuration parameter could be controlled with permissions, allowing just system administrators to change it. But having the ability to change a kernel configuration parameter during runtime which can cause a kernel panic sound like a bad solution. And if either of these is to be implemented the first alternative is favoured.

As the integer version has to be implemented in either case, an implementation of a floating-point version will be ignored at this point.

Integer arithmetic

The need for a integer arithmetic version of the MulTFRC algorithm has been established. A version of that algorithm that achieves this has been contributed by Stein Gjessing. In order to maintain precision during the calculation in the integer version, a lot of scaling has been used to prevent any variable from getting overflow. Even though this works seemingly

good, quick tests shows that the result of the integer version and the floating-point version differs some.

The integer algorithm starts out by scaling up all the input variables. The scales differ depending on the value of the input parameters. The scaling value of each input parameter is stored and is later on used to scale both up and down as the algorithm calculates its way towards the final integer throughput. As a consequence of this scaling the end result differs some from the floating-point version. The percentage of results that differs more than 10% from its counterpart is close to 0.4%, but some of these results might differ as much as 55%. This percentage is the result of more than 12 billion calculations where all input parameters are uniquely put together. To make this differential less between them, several methods has been looked into.

Optimize scaling

The first method of choice to improve the end result was trying to optimize the scaling of the algorithm so that it would become more accurate. This was however not as easy as it might sound. There are so many different values to take in consideration whenever something is scaled up or down. To find points in the algorithm where improvements could be made, parts of the algorithm where isolated to check the differential of its counterpart in the floating-point version. The first point in the algorithm where a differential of interest were observed is at the point where *j* is used as power of exponent *(1-1/N)*. The power of function can be seen in listing 3.17.

Listing 3.17: MulTFRC equation

```
1  u32 mulTFRCPower(u32 N, u32 j, u64 N_CNST, u64 J_CNST) {
2      /*
3          Calculates the scaled version of (1-1/N)^j.
4          Find the non scaled value of j, called jF, and then interpolate either
5          between floor(jF) and floor(jF) + 1/2 (called lower interval) or
6          between floor(jF) + 1/2 and ceil(jF) (called upper interval)
7       */
8
9      u64 jF = j, remainder, tmp;
10     u64 lowResult = 0, middleResult = 0 , upResult = 0;
11     u64 finalResult = 0, diffRes, xSize = 0, ySize = 0, yStep = 0;
12     u64 oneMinus, result;
13     remainder = do_div(jF, J_CNST);
14     oneMinus = N_CNST - N_CNST * N_CNST;
15     result = oneMinus;
16     do_div(oneMinus, N);
17
18     if(jF == 1) {
19         lowResult = oneMinus;
20         upResult = oneMinus * oneMinus;
21         do_div(upResult, N_CNST);
22     } else
23         if(jF == 2) {
24             lowResult = oneMinus * oneMinus;
```

```
25              do_div(lowResult, N_CNST);
26              upResult = result * oneMinus;
27              do_div(upResult, N_CNST);
28          }
29          else {
30              u32 ind;
31              result = oneMinus * oneMinus;
32              for (ind = 2; ind < jF; ind++) {
33                  result = result * oneMinus;
34                  do_div(result, N_CNST);
35              }
36              upResult = result * oneMinus;
37              do_div(upResult, N_CNST);
38              do_div(upResult, N_CNST);
39              lowResult = result;
40              do_div(lowResult, N_CNST);
41          }
42
43      if(remainder == 0)
44          return lowResult;
45
46      middleResult = lowResult * lsqrt(oneMinus * N_CNST);
47      do_div(middleResult, N_CNST);
48      upResult = (lowResult * oneMinus);
49      do_div(upResult, N_CNST);
50
51      tmp = J_CNST;
52      do_div(tmp, 2);
53      if (remainder <= tmp)
54          diffRes = middleResult - lowResult;
55      else
56          diffRes = upResult - middleResult;
57
58      yStep = (diffRes * N_CNST);
59      do_div(yStep, tmp);
60
61      if (remainder > tmp)
62          xSize = remainder - tmp;
63
64      ySize = yStep * xSize;
65      do_div(ySize, N_CNST);
66
67      if (remainder <= tmp)
68          finalResult = lowResult + ySize;
69      else
70          finalResult = middleResult + ySize;
71
72      return finalResult;
73  }
```

Attempts to increase precision for this result did not provide any better solution to the problem than what the original integer version already did.

To create some statistic for every change to see if it was for the better or worse would take too much time for two reasons. Firstly every test run of

the algorithm with 12 billion unique input values takes about twelve hours on the available test machine. And secondly the amount of combinations of values is so great that it is hard to fit in a humanly readable way. After a few weeks of trying to increase precision with improved scaling it was considered too much of a time consumption.

Replacement values

A second approach to the problem was to create a table of replacement values that according to statistics usually did not provide much difference in between the floating version and the integer version. All parameters, except p, comes in so many different values that they are impossible to create a replacement map for them. The p parameter is limited to 1000 different values in the original *TFRC* code within the kernel. A replacement map for less than 1000 values is not considered to be too huge for the kernel. The idea was then to try and replace values of p where the result differed most and replace it with the closest p where it differed less. A program to map to closest value above or below current replacement value was created to run test with this potential solution. The values which were replaced had a result that differed more than 40%. The replacement values was the closest value of p where the result differed less than 40%. This did however result in even higher differentials than before it was introduced.

Checking relevant input

Exclusion of value combinations that would not occur in a real environment is a way to see if the result improves. This is a realistic assumption as high loss rate does not provide a low amount of packets lost. With this in mind the course was set to exclude high values of j combined with low values of p and visa verse. Also all values of p that is not available in the lookup table or above 0.9 were excluded. A scenario where more than 90% of the packets ($p > 0.9$) are lost is not very likely. If however this scenario is relevant for someone, they might not care that much if the algorithm is off by about 50% in a relative low amount of calculations. Secondly the combination of p and j values, which should not occur, were skipped. The amount of results that has more than 10% difference is 0.4% or lower in all cases. Table below shows statistics of the tests.

The first case only exclude high values of p. That is why the total runs are much higher that the other. The second and third case excludes combinations of p and j that seems unlikely to occur. The third also only uses the initial scaled values that the kernel has, while the second lets the algorithm scale them itself.

The table shows that the kernel scaled input values have slightly less occurrences of differentials above 10%. This does not mean anything of the algorithm is changed. It simply shows that in a real and likely scenario the occurrences is actually as low as *0.365%*. It also shows that it is slightly better to use the already scaled values of the kernel than to let the algorithm use different scales depending on the input values. This could not be

Table 3.1: Table of algorithm tests

Approach	Runs	Occurrences	Percentage
No p above 0.9	5630781904	22526307	0.400%
Algorithm scaled input	1178639210	4719621	0.400%
Kernel scaled input	1214926500	4439454	0.365%

considered an improvement of the algorithm. Although the algorithm does save a few calculations because it does not have to find the scaled values itself.

Square root

The only thing remaining before the integer version of the algorithm can be implemented fully is a way to solve square root with only integers. The square root of a number is defined as one out of two equal numbers that when multiplied with each other gives you the original number. The square root of *4* is *2* as *2 * 2 = 4*, and for *5* the square root is *2,236...* As this example shows, only using integers will not provide an accurate result for most inputs. The result will at best be rounded down to the closest square root integer of the input. This algorithm has been tested on all possible input values and the result has proven that it always gives the correct floor value of square root.

Listing 3.18 shows how the square root solution was implemented. It starts out by declaring necessary variables. Variable *scale* is used to scale down input *i* if it is too high. This needs to be done because of the way square root is solved. If for instance input *i* is the highest possible *s64*, the mid value will be half of *i* and then multiplied by itself. The result will then overflow. A few different scales where chosen not to scale down too little or too much. If the number is scaled a square root will be calculated of this scaled value and then the square root will be scaled back up to get the precise square root of the original input *i*.

The main calculation can start When *low* and *high* has been initialized. Firstly a *mid* value has to be picked. This value is either determined by the result of the scaled calculation or as half of *low* + *high*. When *mid* is set, the *midpwr* can be calculated. The calculation of *midpwr* is the point where a too large value of *mid* would make the algorithm overflow.

Listing 3.18: net/dccp/ccids/lib/multfrc_equation.c

```
1  s64 lsqrt(s64 i) {
2      s64 originalI = i, scale = 0;
3      s64 low, high, mid, midpwr, tmp;
4
5      // Need to scale down if high n (n > 12148001997L)
6      if(i > 12148001997ll) {
7          if(i > 12148001997999999999ll)
8              scale = 1000000ll;
```

```
 9          else if(i > 3043978089999999ll)
10              scale = 10000ll;
11          else
12              scale = 1000ll;
13          tmp = scale * scale;
14          do_div(i, tmp);
15          i = lsqrt(i);
16          i += 1;
17          i *= scale;
18      }
19
20      low = 1;
21      high = i;
22      do_div(high, 2);
23      if(scale) {
24          mid = i;
25      } else {
26          mid = low + high;
27          do_div(mid, 2);
28      }
29      midpwr = mid * mid;
30
31      while ((low < (high - 1)) && midpwr != originall) {
32          if (midpwr < originall && midpwr > 0) {
33              low = mid;
34          } else {
35              high = mid;
36          }
37          mid = low + high;
38          do_div(mid, 2);
39          midpwr = mid * mid;
40      }
41
42      return mid;
43  }
```

Square root example

If the input to the algorithm is *i = 10* the result will be *3*. With floating it would be *3.1623* with a 4 digits precision. If the result had been above *3.5* it would still be rounded down to *3*. This is because the square result should not be greater than the input. If the input *10* is applied to the algorithm it will not be scaled. Variable *high* would be *10* (Line 21 in listing 3.18), and then divided by *2* with *do_div(high, 2)* (Line 22). Variable *mid* would be *low + high = 10* (Line 26) and then divided by *2* with *do_div(mid, 2)* (Line 27). Variable *midpwr* would be *9* (Line 29). First time in the while loop test will return true because *1 (low)* is less than *5 (high) - 1 = 4* and *9 (midpwr)* is not equal to *10 (input)* (Line 31). Although this is what the end result will be, the algorithm needs to check if a higher value of square root can be used to get closer to input, but still as a lower value than the input. The if test will return true in this while loop with the conditions *9 (midpwr)* is less than *10*

(input) and *9 (midpwr)* is greater than *0*. Therefore the value of *low* is now set to *3 (mid)* (Line 33). A new value of *mid* is now calculated to be *3 (low) + 5 (high)* (Line 37), and then divided by *2 (do_div(mid, 2))* resulting in a value of *4* (Line 38). *midpwr* is now the square of *4 (mid)* which is *16* (Line 39). The while loop is not hit for the second time and still returns true. *3 (low)* is less than *5 (high) - 1* and *16 (midpwr)* is not equal to *10 (input)* (Line 31). The if test returns false this time. The first condition is false with the statement *16 (midpwr)* is greater than *10 (input)* (Line 32). The else block is than executed. *high* is now assigned the value of *mid* which is *4* (Line 35). *mid* is now assigned the value *3 (low) + 4 (high) = 7* (Line 37). It is than divided by *2* with *do_div(mid, 2)* (Line 38). As the result of this would be *3.5*, the fact that this is not a floating point makes it *3* instead. *midpwr* is again calculated to be *3 * 3 = 9* (Line 39). The while loop is now checked for the third and last time. It returns false with the statement *3 (low)* is less than *4 (high) - 1*. As the first condition of the while test is false, it is guaranteed that the closest value of a square to input, that is still less than input, has been found. This is a fact as there are no more numbers to test between *low* and *high*. The value *3 (mid)* is now the best match of a square root and is returned.

MulTFRC equation

Listing 3.19 shows the full implementation of the integer version of MulTFRC in the Linux kernel. It is done in the file *net/dccp/ccids/lib/multfrc_equation.c*. The comments on line *1 - 9* states what the input variables is scaled by in the kernel. All input parameters are written to the kernel log before the algorithm starts calculating. The end result is also logged.

Line *84 - 90* handles the first *if* test of listing 3.15. On line 85 the power of function is called. Line *92 - 102* calculates line *7* of listing 3.15. On line 122 the square root function is called. The rest of the algorithm does normal calculations according to the specified nature of listing 3.15. The reason why it is so much longer is that it needs to check how the next operand should be done with the given scale not to overflow and to keep the precision correct. *do_div* is used because it provides a way to divide *64* bit variables on a *32* bit machine, where as a regular / operand would fail.

Listing 3.19: MulTFRC equation

```
1  /**
2   * multfrc_calc_x — Calculate the send rate
3   * @s: packet size in bytes
4   * @R: RTT scaled by 1000000 (i.e., microseconds)
5   * @p: loss ratio estimate scaled by 1000000
6   * @N: cumulative flows scaled by 10000
7   * @j: losses per event scaled by 10000
8   * Returns X_ calc in bytes per second (not scaled).
9   */
10 u32 multfrc_calc_x(u16 s, u32 R, u32 p, u32 N, u32 j) {
11     s64 P_CNST = 1000000;
12     s64 R_CNST = 1000000;
13     s64 N_CNST = 10000;
```

```
14      s64 ROOT_N_CNST = 100; // Root of N_CNST
15      s64 J_CNST = N_CNST; // Must be same as N_CNST in order to compare j and n
16      s64 maxValue = 4294967295ul; // Max value of u32
17      s64 maxVdivJ_CNST, maxVdivN_CNST, maxVdivR_CNST, maxVdivP_CNST;
18
19      s64 t_RTO;
20      s64 b = 1;
21      s64 X_calc = 0;
22
23      u64 x;
24      s64 af;
25      s64 z;
26      s64 q;
27      s64 afN;
28      s64 a1;
29      s64 a2;
30      s64 r1a;
31      s64 r2a;
32      s64 x1;
33      s64 x2;
34      s64 z1;
35      s64 m1;
36      s64 m2;
37      s64 m3;
38      s64 m4;
39      s64 w1 = 0;
40      s64 w2 = 0;
41      s64 w3 = 0;
42      s64 w4 = 0;
43
44      s64 a10;
45      s64 a11;
46      s64 a13;
47      s64 a14;
48      s64 a15;
49      s64 x10;
50      s64 x11;
51      s64 x20;
52      s64 x21;
53      s64 z10;
54      s64 z11;
55      s64 m10;
56      s64 m11;
57      s64 m30;
58      s64 m31;
59      s64 m32;
60      s64 m40;
61      s64 m41;
62      s64 m43;
63      s64 w20;
64      s64 w21;
65      s64 w22;
66      s64 w30;
67
```

```
68      s64 ceilN;
69      s64 intPart;
70
71      s64 CONST;
72
73      s64 w3A;
74      s64 w3B;
75      s64 w3Bi;
76      s64 factorW;
77
78      s64 tmp;
79
80      t_RTO = R * 4;
81
82      printk("MulTFRC input s: %d, R: %d, p: %d, N: %d, j: %d\n", s, R, p, N, j);
83
84      if (N > 1 * N_CNST && N < 12 * N_CNST && j < N * 5) {
85          af = (N + 1) * (1 * N_CNST - mulTFRCPower(N, j, N_CNST, J_CNST));
86          do_div(af, N_CNST);
87      } else {
88          af = j * N_CNST;
89          do_div(af, J_CNST);
90      }
91
92      intPart = N;
93      do_div(intPart, N_CNST);
94      if(N < N_CNST) {
95          ceilN = N_CNST;
96      } else if (intPart * N_CNST == N) {
97          ceilN = N;
98      } else {
99          ceilN = (intPart + 1) * N_CNST;
100     }
101
102     af = max(min(af, ceilN), N_CNST);
103
104     afN = (af * N_CNST);
105     do_div(afN, N);
106
107     a10 = b * af;
108     a11 = 2 * afN;
109     a13 = N_CNST - a11;
110     a14 = a10 * a13;
111     a15 = a14 * a13;
112     a1 = b * af * (1 * N_CNST - 2 * afN) * (1 * N_CNST - 2 * afN);
113     tmp = N_CNST * N_CNST;
114     do_div(a1, tmp);
115
116     a2 = 24 * P_CNST;
117     do_div(a2, p);
118     tmp = a1;
119     do_div(tmp, N_CNST);
120     a2 += tmp;
121
```

```
122     r1a = lsqrt(b * af);
123
124     a2 = a2 * N_CNST;
125     r2a = lsqrt(a2);
126
127     x10 = b * afN;
128     x11 = x10 * (2 * af - N);
129     x1 = b * afN * (2 * af - N);
130
131     x20 = r1a * r2a;
132     x21 = x20 * (ROOT_N_CNST * N_CNST);
133     x2 = x1 + (r1a * r2a * N_CNST);
134
135     tmp = 6 * N;
136     x = x2;
137     do_div(x, tmp);
138
139     z10 = (32 * p) * p;
140
141     z11 = (P_CNST * P_CNST) + z10;
142     z1 = (P_CNST * P_CNST + (32 * p * p));
143     do_div(z1, P_CNST);
144     z1 *= t_RTO;
145
146     z = z1;
147     tmp = (P_CNST - p);
148     do_div(z, tmp);
149
150     m10 = b * N_CNST;
151     m11 = m10 * N_CNST;
152     m1 = b * N_CNST * N_CNST * N_CNST;
153     do_div(m1, x);
154
155     if (N_CNST + 3 * N * J_CNST > 1024 * j) {
156         m2 = N_CNST + 3 * N * J_CNST;
157         do_div(m2, j);
158         m2 *= x;
159     } else {
160         m2 = 3 * N * J_CNST;
161         tmp = j * N_CNST;
162         do_div(m2, tmp);
163         m2 = (1 + m2) * x * N_CNST;
164     }
165
166     m30 = 2 * j * z;
167     m31 = m1 * N_CNST;
168     m32 = R * m2;
169     tmp = maxValue;
170     do_div(tmp, m1);
171     if((2 * j * z) < tmp) {
172         m3 = 2 * j * z * m1;
173         tmp = R * m2;
174         do_div(m3, tmp);
175         m3 *= N_CNST;
```

```
176     } else {
177         m3 = 2 * j * z;
178         do_div(m3, m2);
179         m3 *= m1 * N_CNST;
180         do_div(m3, R);
181     }
182
183     m4 = 0;
184     m40 = N * z;
185     m41 = m40 * N_CNST;
186     m43 = x * R;
187     maxVdivR_CNST = maxValue;
188     do_div(maxVdivR_CNST, R_CNST);
189     if (N * z * N_CNST < maxVdivR_CNST) {
190         m4 = N * z * N_CNST * R_CNST;
191         tmp = x * R;
192         do_div(m4, tmp);
193     } else {
194         u32 factor = 64;
195         u32 dividend = N * z * N_CNST;
196         do_div(dividend, 64);
197         while (dividend > maxVdivR_CNST) {
198             do_div(dividend, 64);
199             factor = factor * 64;
200         }
201         m4 = dividend * R_CNST;
202         tmp = x * R;
203         do_div(m4, tmp);
204         m4 *= factor;
205     }
206
207     maxVdivJ_CNST = maxValue;
208     do_div(maxVdivJ_CNST, J_CNST);
209     if (m4 > maxVdivJ_CNST && m3 > maxVdivR_CNST) {
210         q = N * R_CNST * J_CNST;
211     } else {
212         if (m4 > maxVdivJ_CNST) {
213             q = min(m3 * R_CNST, N * R_CNST * J_CNST);
214         } else if (m3 > maxVdivR_CNST) {
215             q = min(m4 * J_CNST, N * R_CNST * J_CNST);
216         } else {
217             q = min(min(m3 * R_CNST, m4 * J_CNST), N * R_CNST * J_CNST);
218         }
219     }
220
221     if(q < 0)
222         DCCP_CRIT("ERROR: q LESS THAN 0");
223
224     tmp = q;
225     w1 = ((R_CNST * J_CNST) - q);
226     do_div(w1, N);
227
228     CONST = P_CNST;
229     do_div(CONST, 100);
```

```
230
231     if(w1 == 0) {
232         w3 = 0;
233     } else {
234         w20 = p * x;
235
236         if (p * x > 512 * P_CNST) {
237             w21 = w20;
238             do_div(w21, P_CNST);
239             w21 *= R;
240             w2 = p * x;
241             do_div(w2, P_CNST);
242             w2 *= R;
243
244             w30 = w21 * N;
245             w3A = N_CNST * R_CNST * J_CNST;
246             do_div(w3A, w2);
247         } else {
248             if (p * x > 512 * CONST) {
249                 w21 = p * x;
250                 do_div(w21, CONST);
251                 w21 *= R;
252                 w2 = p * x;
253                 do_div(w2, CONST);
254                 w2 *= R;
255                 do_div(w2, 100);
256
257                 w30 = w21 * N;
258                 w3A = N_CNST * R_CNST * J_CNST;
259                 do_div(w3A, w2);
260             } else {
261                 w22 = p * x * R;
262                 w2 = p * x * R;
263                 w30 = w2 * 512;
264
265                 maxVdivN_CNST = maxValue;
266                 do_div(maxVdivN_CNST, N_CNST);
267                 if (P_CNST * R_CNST > maxVdivN_CNST) {
268                     if(P_CNST * R_CNST > w2 * 512) {
269                         w3A = P_CNST * R_CNST;
270                         do_div(w3A, w2);
271                         w3A *= J_CNST * N_CNST;
272                     } else{
273                         w3A = P_CNST * R_CNST;
274                         tmp = w2;
275                         do_div(tmp, N_CNST);
276                         do_div(w3A, tmp);
277                         w3A *= J_CNST;
278                     }
279                 } else {
280                     w3A = N_CNST * R_CNST * P_CNST;
281                     do_div(w3A, w2);
282                     w3A *= J_CNST;
283                 }
```

```
284             }
285         }
286
287         factorW = 1;
288         if (q == 0) {
289             w3B = 0;
290         } else {
291             tmp = maxValue;
292             do_div(tmp, q);
293             if (w3A < tmp) {
294                 w3B = w3A * q;
295                 tmp = N * R_CNST * J_CNST;
296                 do_div(w3B, tmp);
297             } else {
298                 tmp = maxValue;
299                 if (q > w3A) {
300                     u32 qA = q;
301                     do_div(tmp, qA);
302                     while (w3A > tmp) {
303                         do_div(qA, 100);
304                         factorW = factorW * 100;
305                     }
306                     w3Bi = w3A * qA;
307                 } else {
308                     u32 wA = w3A;
309                     do_div(tmp, wA);
310                     while (q > tmp) {
311                         do_div(wA, 100);
312                         factorW = factorW * 100;
313                     }
314                     w3Bi = q * wA;
315                 }
316
317                 if (J_CNST > factorW) {
318                     w3B = w3Bi;
319                     do_div(w3B, J_CNST);
320                     w3B *= factorW;
321                     tmp = R_CNST * N;
322                     do_div(w3B, tmp);
323                 } else if (N > factorW) {
324                     w3B = w3Bi;
325                     do_div(w3B, N);
326                     w3B *= factorW;
327                     tmp = R_CNST * J_CNST;
328                     do_div(w3B, tmp);
329                 } else if (R_CNST > factorW) {
330                     w3B = w3Bi;
331                     do_div(w3B, R_CNST);
332                     w3B *= factorW;
333                     tmp = J_CNST * N;
334                     do_div(w3B, tmp);
335                 } else if (J_CNST * N > factorW) {
336                     w3B = w3Bi;
337                     tmp = J_CNST * N;
```

```
338                     do_div(w3B, tmp);
339                     w3B *= factorW;
340                     do_div(w3B, R_CNST);
341                 } else if (J_CNST * R_CNST > factorW) {
342                     w3B = w3Bi;
343                     tmp = J_CNST * R_CNST;
344                     do_div(w3B, tmp);
345                     w3B *= factorW;
346                     do_div(w3B, N);
347                 } else if (N * R_CNST > factorW) {
348                     w3B = w3Bi;
349                     tmp = N * R_CNST;
350                     do_div(w3B, tmp);
351                     w3B *= factorW;
352                     do_div(w3B, J_CNST);
353                 } else {
354                     w3B = w3Bi;
355                     tmp = J_CNST * N * R_CNST;
356                     do_div(w3B, tmp);
357                     w3B *= factorW;
358                 }
359             }
360         }
361         w3 = w3A - w3B;
362     }
363
364     maxVdivP_CNST = maxValue;
365     do_div(maxVdivP_CNST, P_CNST);
366     if (q > N_CNST * R_CNST * J_CNST || q > maxVdivP_CNST) {
367         w4 = q;
368         tmp = z * (P_CNST - p);
369         do_div(w4, tmp);
370
371         if (((w3 * N_CNST) + (w4 * P_CNST)) > (1024 * N_CNST * J_CNST)) {
372             X_calc = (w3 * N_CNST) + w4 * P_CNST;
373             tmp = N_CNST * J_CNST;
374             do_div(X_calc, tmp);
375             X_calc *= s;
376         } else {
377             X_calc = (w3 * N_CNST + w4 * P_CNST) * s;
378             tmp = N_CNST * J_CNST;
379             do_div(X_calc, tmp);
380         }
381     } else {
382         w4 = q * P_CNST;
383         tmp = z * (P_CNST - p);
384         do_div(w4, tmp);
385
386         if ((w3 * N_CNST + w4) > (1024 * N_CNST * J_CNST)) {
387             X_calc = (w3 * N_CNST) + w4;
388             tmp = N_CNST * J_CNST;
389             do_div(X_calc, tmp);
390             X_calc *= s;
391         } else {
```

```
392             X_calc = ((w3 * N_CNST) + w4) * s;
393             tmp = N_CNST * J_CNST;
394             do_div(X_calc, tmp);
395         }
396     }
397
398     printk("MulTFRC x_calc: %d\n", X_calc);
399     return X_calc;
400 }
```

After copy: Scale of input Add CCID 5 to net/dccp/probe.c added htcx %d and hctx->ccid5hctx_j Add CCID 5 multfrc j in ccid.h net/dccp/ccids/lib/mulloss_interval.c added loss count from rfc Add multfrc init in net/dccp/ccid.c Add MulTFRC and remove CCID5-TFRC test in Kconfig Explain do_div

3.2.2 Changes to section 4 of RFC 5348

The procedure for updating the allowed sending rate is replaced with an if test. If p is equal to 1, it will not calculate the sending rate with the MulTFRC algorithm, but rather with a simple operation shown in listing 3.20 on line 3. (p is scaled but 1000000 in the kernel) The calculation is packet size (*hctx->ccid5hctx_s*) multiplied with N divided by the back off value set for MulTFRC (*MULTFRC_T_MBI*). The back off value is defined in seconds.

```
#define MULTFRC_T_MBI 64
```

Listing 3.20: net/dccp/ccids/ccid5.c

```
1   /* Update sending rate (step 4 of [RFC 3448, 4.3]) */
2   if (hctx->ccid5hctx_p == 1000000) {
3       hctx->ccid5hctx_x_calc = hctx->ccid5hctx_s * N / MULTFRC_T_MBI;
4   } else if (hctx->ccid5hctx_p > 0) {
5       hctx->ccid5hctx_x_calc =
6       multfrc_calc_x(hctx->ccid5hctx_s,
7               hctx->ccid5hctx_rtt,
8               hctx->ccid5hctx_p, N, hctx->ccid5hctx_j);
9   }
10  ccid5_hc_tx_update_x(sk, &now);
```

3.2.3 Changes to section 5 of RFC 5348

This section of changes concerns the calculation of j, the average number of packets lost in a loss event. The calculation is added into the file *net/dccp/ccids/lib/mulloss_interval.c* as an addition to the code responsible of finding p. A few variable declarations has to be added along with the code seen in listing 3.21. All code before line number 21 is dedicated to finding p. As MulTFRC also requires j to be calculated, the operations for this is shown below line 21. It is recommended in the specification of "MulTFRC: TFRC with weighted fairness" [8] that the eight last loss

intervals should be used for this calculation. The array size of the loss history events is precisely eight and the calculation is therefore following this recommendation. The length of the array is fetched with *multfrc_lh_length(lh)* and stored to variable *k*. The size is decreased by 1 as the for loops using *k* start at 0. The value of *j* is scaled down to the same scale used for *N* in the MulTFRC algorithm.

Listing 3.21: net/dccp/ccids/lib/mulloss_interval.c

```
1  static void multfrc_lh_calc_i_mean(struct multfrc_loss_hist *lh)
2  {
3      u32 i_i, i_tot0 = 0, i_tot1 = 0, w_tot = 0, LP_tot = 0;
4      int i, k = multfrc_lh_length(lh) - 1; /* k is as in rfc3448bis, 5.4 */
5
6      if (k <= 0)
7          return;
8
9      for (i = 0; i <= k; i++) {
10         i_i = multfrc_lh_get_interval(lh, i);
11
12         if (i < k) {
13             i_tot0 += i_i * multfrc_lh_weights[i];
14             w_tot += multfrc_lh_weights[i];
15         }
16         if (i > 0)
17             i_tot1 += i_i * multfrc_lh_weights[i-1];
18     }
19
20     lh->i_mean = max(i_tot0, i_tot1) / w_tot;
21
22     if (i_tot0 > i_tot1) {
23         for (i = 0; i < k; i++) {
24             LP_tot += (multfrc_lh_get_interval(lh, i) * multfrc_lh_weights[i]);
25         }
26     } else {
27         for (i = 0; i <= k; i++) {
28             if (i < k) {
29                 LP_tot += (multfrc_lh_get_interval(lh, i) * multfrc_lh_weights[i]);
30             } else {
31                 LP_tot += (multfrc_lh_get_interval(lh, i) * multfrc_lh_weights[i-1]);
32             }
33         }
34     }
35
36     /* Scaling down to 10000 for MulTFRC algorithm. Needs to be same scale as N. */
37     lh->j = LP_tot / w_tot / 100;
38 }
```

Section 5.5 of RFC 5348 not implemented

Changes to section 5.5 of RFC 5348 [5] are also mentioned in "MulTFRC: TFRC with weighted fairness" [8]. This section is however an optional part

of the RFC and has not been implemented in the kernel for this reason it. Thus it is not implemented as a part of this thesis either.

3.2.4 Changes to section 6 of RFC 5348

When *p* and *j* is calculated, the previous value of both are to be preserved. This is already done for *p*. In listing 3.22 the value of *p* is represented as *i_mean* and its previous value is stored as *old_i_mean*. The preservation of *j* will be done in the same manner. Thus it is stored as *old_j*.

Listing 3.22: net/dccp/ccids/lib/mulloss_interval.c

```
1  u8 multfrc_lh_update_i_mean(struct multfrc_loss_hist *lh, struct sk_buff *skb)
2  {
3      struct multfrc_loss_interval *cur = multfrc_lh_peek(lh);
4      u32 old_i_mean = lh->i_mean;
5      u32 old_j = lh->j;
6      s64 len;
```

When a packet is received and no loss history is initialised it means that this is the first received packet and according to the specifications of the draft [8] *j* should now be set to 0. If this does not occur *multfrc_lh_update_i_mean* calculates a new *p* and *j*. If *p* has increased a feedback parameter is set and a subsequent feedback control packet is sent.

Listing 3.23: net/dccp/ccids/ccid5.c

```
1      if (!multfrc_lh_is_initialised(&hcrx->ccid5hcrx_li_hist)) {
2          const u32 sample = multfrc_rx_hist_sample_rtt(&hcrx->ccid5hcrx_hist, skb);
3
4          /*
5           * Set j to 0 when loss history is initialized.
6           */
7          hcrx->ccid5hcrx_j = 0;
8
9          /*
10          * Empty loss history: no loss so far, hence p stays 0.
11          * Sample RTT values, since an RTT estimate is required for the
12          * computation of p when the first loss occurs; RFC 3448, 6.3.1.
13          */
14         if (sample != 0)
15             hcrx->ccid5hcrx_rtt = multfrc_ewma(hcrx->ccid5hcrx_rtt, sample, 9);
16
17     } else if (multfrc_lh_update_i_mean(&hcrx->ccid5hcrx_li_hist, skb)) {
18         /*
19          * Step (3) of [RFC 3448, 6.1]: Recompute I_mean and j, if I_mean
20          * has decreased (resp. p has increased), send feedback now.
21          */
22         do_feedback = CCID5_FBACK_PARAM_CHANGE;
23     }
```

When the first loss interval occurs *j* is set to 1.

Listing 3.24: net/dccp/ccids/ccid5.c

```
1  static u32 ccid5_first_li(struct sock *sk)
2  {
3      struct ccid5_hc_rx_sock *hcrx = ccid5_hc_rx_sk(sk);
4      u32 x_recv, p, delta;
5      u64 fval;
6
7      /*
8       * Set j to 1 when first loss interval occur
9       */
10     hcrx->ccid5hcrx_j = 1;
```

3.2.5 Setting N

The value of *N* must be set before the transmission of data packets can start. Preferably it is set with the creation of sockets on each side of a connection. Both sender and receiver have to set a value of *N* each. As this value might not be equal, the lowest value should be chosen as the value of *N* used for the connection. Setting this value with sockets requires patching of test programs like *iperf* to use other than default value provided by the socket. This is potentially time consuming and for some other test programs that exists it might not be an option to patch it in. As a solution for running tests with already existing programs, the value must be set elsewhere for this purpose.

Setting N with sysctl

Creating a *sysctl* value for *N* is an easy way to accomplish this. Setting a *sysctl* value does not fully provide the functionality required. It will still lack the ability to exchange and choose the lowest value of *N* for the session. For the sake of simplicity this part will not be implemented. Testing of the implementation will be within a controlled environment as far as configuration is concerned. A guarantee that both sender and receiver will use the same value of *N* can therefore be given. It would also not have any effect one the testing of the algorithm even if it were to be implemented. To create a *sysctl* variable for *N* it has to be set via the *net/dccp/sysctl.c* and available in *net/dccp/ccids/ccid5.c*.

First a global *u32 N* were created in *net/dccp/ccids/ccid5.c*. See listing 3.25. The *__read_mostly* is present to inform the compiler that this variable will not be changed very often. The compiler will then do optimizations with these kind of variables that makes them quicker to read with some form of caching, thus writes to the variables is slower. The variable is also given a default value of *N = 1*, which when scaled up to kernel normalized standard is *1000000*.

Listing 3.25: net/dccp/ccids/ccid5.c

```
1  u32 N __read_mostly = 10000;
```

Variable *N* needs to be available for *net/dccp/sysctl.c*. In order to achieve this an *extern u32 N* were created in *net/dccp/ccids/ccid5.h*. With an include

of *net/dccp/ccids/ccid5.h* in *net/dccp/sysctl.c* the variable *N* is now available for *sysctl*.

Listing 3.26: net/dccp/ccids/ccid5.h

```
1 extern u32 N;
```

A *sysctl* variable to control *N* is only needed when the kernel is compiled with CCID 5. *#ifdef* should therefore be used to only include the blocks of code that concerns this *sysctl* variable when CCID 5 is enabled. The values of *long_one* and *long_max* are used as available range of choices for *N*, former being minimum and latter being maximum. (*0UL* is max value of a *unsigned long*) See listing 3.27.

Listing 3.27: net/dccp/sysctl.c

```
1 #ifdef CONFIG_IP_DCCP_CCID5
2 #include "ccids/ccid5.h"
3 static unsigned long long_one = 1;
4 static unsigned long long_max = ~0UL;
5 #endif
```

Further down in *net/dccp/sysctl.c* there is an array of structs containing all *sysctl* options in the *dccp_default* namespace. These have a prefix of *net.dccp.default*. In listing 3.28 a *static struct ctl_table* is set up for *N*. The *.procname* is *multtfrc_n* which gives *net.dccp.default.multfrc_n* as full name in *sysctl*. *.data* indicates where the data of this kernel parameters will be stored. In this case *N*, included from *net/dccp/ccids/ccid5.h*. *.maxlen* defines the maximum size of data to be stored. In this case size of *N* is used. *.mode* specifies permissions for the file stored for this configuration parameter. It is keep equal to the rest of the configuration parameters already implemented for DCCP. The *.proc_handler* handles the text input of the *sysctl*. In this case a *proc_doulongvec_minmax* is used. It will read an unsigned long with a minimum and maximum value. The *.extra* values are options provided to the *.proc_handler*. And in this case they are provided as the minimum and maximum value for *N*. Added in: static struct ctl_table dccp_default_table[]

Listing 3.28: net/dccp/sysctl.c

```
1  #ifdef CONFIG_IP_DCCP_CCID5
2      {
3          .procname = "multfrc_n",
4          .data = &N,
5          .maxlen = sizeof(N),
6          .mode = 0644,
7          .proc_handler = &proc_doulongvec_minmax,
8          .extra1 = &long_one,
9          .extra2 = &long_max,
10     },
11 #endif
```

The kernel configuration parameter for *N* can now be set with the following command:

$ sysctl -w net.dccp.default.multfrc_n=10000

3.2.6 Additional changes

As a consequence of the previous additions and changes to the kernel there are also more alterations that needs to be made. Already existing data structures needs to reflect recent additions of variables.

The file *include/linux/multfrc.h* was added to provide both sender and receiver with the data structure needed. These structures are used to send the algorithm variables between the transceivers. Aside from MulTFRC having its own data structures, the most important difference from the TFRC data structure is the addition of *j*. MulTFRC can now send the value of *j* back and forth. The data structures can be view in listing 3.29.

Listing 3.29: include/linux/multfrc.h

```
1  #ifndef _LINUX_MULTFRC_H_
2  #define _LINUX_MULTFRC_H_
3
4  #include <linux/types.h>
5
6  struct multfrc_rx_info {
7          __u32 multfrcrx_x_recv;
8          __u32 multfrcrx_rtt;
9          __u32 multfrcrx_p;
10         __u32 multfrcrx_j;
11 };
12
13 struct multfrc_tx_info {
14         __u64 multfrctx_x;
15         __u64 multfrctx_x_recv;
16         __u32 multfrctx_x_calc;
17         __u32 multfrctx_rtt;
18         __u32 multfrctx_p;
19         __u32 multfrctx_rto;
20         __u32 multfrctx_ipi;
21         __u32 multfrctx_j;
22 };
23
24 #endif /* _LINUX_MULTFRC_H_ */
```

To make sure the received value of *j* does not exceed the allowed value the check of listing 3.30 were added. The same is also done with the value of *p*.

Listing 3.30: net/dccp/ccids/ccid5.c

```
1      j = opt_recv->ccid5or_j;
2          if (j == ~0U || j == 0)
3                  hctx->ccid5hctx_j = 0;
4          else /* can not exceed 100% */
5                  hctx->ccid5hctx_j = j;
```

An initialization function was made for the MulTFRC library in section 3.1.6 in the file *net/dccp/ccids/lib/multfrc.c*. This library does exactly the same as for TFRC. This includes initialization off loss interval and packet history. These are also included in the tear down. To make sure the MulTFRC library is initialized, a pointer to the available CCID operations is added for CCID 5 in listing 3.31 on line *8 - 10*. The library is then initialized when the CCIDs are activated. The initialization call for MulTFRC can be seen on line *22* and the corresponding tear down on line *47*.

Listing 3.31: net/dccp/ccid.c

```
1  #include "ccids/lib/multfrc.h"
2
3  static struct ccid_operations *ccids[] = {
4          &ccid2_ops,
5  #ifdef CONFIG_IP_DCCP_CCID3
6          &ccid3_ops,
7  #endif
8  #ifdef CONFIG_IP_DCCP_CCID5
9          &ccid5_ops,
10 #endif
11 };
12
13 [...]
14
15 int __init ccid_initialize_builtins(void)
16 {
17         int i, err = tfrc_lib_init();
18
19         if (err)
20                 return err;
21
22         if (err = multfrc_lib_init())
23                 return err;
24
25         for (i = 0; i < ARRAY_SIZE(ccids); i++) {
26                 err = ccid_activate(ccids[i]);
27                 if (err)
28                         goto unwind_registrations;
29         }
30         return 0;
31
32 unwind_registrations:
33         while(--i >= 0)
34                 ccid_deactivate(ccids[i]);
35         tfrc_lib_exit();
36         multfrc_lib_exit();
37         return err;
38 }
39
40 void ccid_cleanup_builtins(void)
41 {
42         int i;
43
```

```
44          for (i = 0; i < ARRAY_SIZE(ccids); i++)
45                  ccid_deactivate(ccids[i]);
46          tfrc_lib_exit();
47          multfrc_lib_exit();
48  }
```

Chapter 4

Tools for testing

There are many tools available for testing a Linux installation in a network. Most of them are not natively made to work with DCCP. And some of the tools does not have to know anything about what happens in the transfer protocols. In the next few sections a brief introduction of the tools used to run tests and evaluate the implementation will be given.

4.1 New machine script

There is a lot of things that needs to be installed and configured when a new machine are to be used for testing. An automated script were made for the purpose of easing the installation and assuring everything is done correctly every time a new machine is introduced.

The script first installs every packet needed from the Debian repository. Afterwards it extracts the kernel and applies all implemented changes. It also fixes a bug in one of the Makefiles of the kernel that would otherwise abort the compilation. The script automatically applies all configuration that manually can be applied with *make menuconfig*. The kernel is then compiled and installed on the new machine. All tools that can not be fetched from the Debian repository are downloaded from custom sites and installed after the kernel compilation is completed. Some also needs to be patched with a DCCP extension. Lastly the script reboots the machine and the machine reboots into the newly compiled kernel.

The network set up is done manually when the new machine script has done its job. Any rules concerning networking in */etc/udev/rules.d/* must be cleared in order for the network configurations added in */etc/network/interfaces* to work.

4.2 Iperf

A commonly used tool to create data streams for the purpose of testing throughput is Iperf. It offers a lot of useful features to test throughput in a network environment. Originally it only supports testing with TCP and UDP, but there exists a patched version that also supports DCCP. The

patched version is needed in this case and the new machine script builds and installs it on all machines used for testing.

Every running instance of Iperf can only use one protocol. But any number of flows can be configured to run coherently from one instance. To test MulTFRC together with any amount of TCP flows two instances has to run on each endpoint. One endpoint will be the server, in this case the sink. The server receives all data it can from the client and sends expected responses. The parameter needed to make the instance a server is *-s*. To make the instance of the server receive DCCP, the parameter *-d* is also needed. TCP is default thus it does not need any parameter. Optionally a port parameter can be used if this eases filtering when monitoring the network traffic.

The client does all the sending of data. This is the side of the connection that needs to specify most parameters to acquire the desired state. The parameter for DCCP (*-d*) must be given to connect to the DCCP sink. Again, no such parameter is needed for the TCP client. Obviously the location of the server has to be given. E.g. *-c 10.0.1.2*. Transmitting time can be set with *-t <seconds>* and is useful to ensure all tests run an equal amount of time. The data sent with each packet can be set with for instance *-l 1400B*. This value were chosen to ensure that no packets are split up by any entity on the network. The last parameters used for testing are the number of flows each instance will use. This parameters were only used for TCP and TFRC as MulTFRC is suppose to handle this by by the use of the *N* variable. The parameter for number of flows are *-P <number of parallel flows>*.

4.3 Tcpdump

Tcpdump is used to capture packets sent or received in a network by the computer doing the monitoring. Analysis of the captured packets can then be made to assess the performance of the network and its entities. It does however not support DCCP packet capturing in the same manner as TCP. The information gained from built in features is limited for DCCP. Exporting it to a readable format and parsing it with self made programs were used as a solution.

Command to capture packets:

$ tcpdump -i eth0 net 10.0.1 -s 100 -w capture.pcap

This command capture packets on device *eth0* where packets are using network with prefix *10.0.1*. It also limits the stored information to the header as it only stores 100 bytes per packet.

Command to port content of pcap file to readable text file:

$ tcpdump -r capture.pcap > capture.txt

There exists a DCCP patch for Tcpdump but it did not yield any different result or needed opportunities than what was already given by the default build.

4.3.1 Tcpdump parsing

As none of the existing programs for parsing pcap files can extract the output of the MulTFRC implementation, a program to do this and output useful information is needed. A program to parse the data from the readable text file was therefore made. The output of this program can be directly plotted with Gnuplot. An average line representing a DCCP data packet in the readable text file can be seen below.

18:57:38.301132 IP 10.0.0.24.49396 > 10.0.1.2.5001: data

The size of data in a data packet is defined by Iperf. The program calculates the amount of data packets per chosen interval of time and sums it up by multiplying each data packet with the packet size defined by Iperf. The time interval chosen to sum up depends on the size of the graph. Iperf gives total throughput and throughput per second as feedback to the user after each completed test. These values were used to confirm that the parsing program is calculating the text file correctly.

4.4 Network emulator

There will be one router and two endpoints in the test environment. The router is responsible for emulating scenarios which resembles networks with different loss frequency while bandwidth and delay stays the same. Linux Traffic Control (*tc*) and Qdisc will be used to lower the bandwidth and increase the delay experienced by the devices on the router. A bandwidth of 10 Mbit and a delay of 100 ms will be used for all testing of MulTFRC in the test environment. The following command can be used to set this environment. And the same must be done on the outgoing interface as well.

$ tc qdisc add dev eth0 handle 1: root tbf rate 10mbit burst 15000 limit 15000

$ tc qdisc add dev eth0 parent 1:1 handle 10: netem delay 50ms loss 0.25%

The Token Bucket Filter (*tbf*) is a queueing discipline that is used to enforce bandwidth. NetEm (*netem*) allows tc to add delay and loss, among other things, on an outgoing device.

The configuration can be purged by the following command:

$ tc qdisc del dev eth0 root

4.5 Test scripts

Running tests is both boring and time consuming if not automated. Scripts were therefore made to run about 50 tests of 5 minutes each. A break of 5 minutes were put in between each test to make sure everything is

cleaned up. The time of each break might be a bit long, but as the tests are automated mostly ran at night this does not matter. The acting router were used as a control node for this script. It could send out commands with no artificial delay or loss. The commands are run after a certain time on the end points. This time were chosen so that the router could switch to any given scenario before the test started. As the test on the clients are also time based, the router knows when it can purge the artificial delay and loss configuration in order to clean up if necessary and then start new tests.

Chapter 5

Testing

Several tests has to be done to evaluate the implementation of MulTFRC. These tests were done with both virtual and physical machines. In a virtual environment the occurrence of unknown factors could be introduced, especially concerning networking. Running tests in a real environment is therefore very important.

A scenario which is created to closely act like an real environment is called a network emulation. This means you can use computers to create flows in between them with a network emulator in between them to introduce a configured scenario. The emulator scenario may include things like bandwidth limits, delay and loss rate. Emulation offers a nice way to assess the performance of systems under various conditions.

5.1 Implementation testing

This section will cover most of the testing that has been done during the course of this thesis. MulTFRC will be tested alone and along with TCP to assess how it behaves. TFRC will also be tested to compare it with MulTFRC. All monitoring and capturing of packets were done on the endpoints and not the router. There is no guarantee whether it will add delay and loss after or before the packets are observed or captured. It is therefore safer to monitor and capture data on the machines who has no knowledge about the interference.

5.1.1 MulTFRC algorithm tests

There are parts of the MulTFRC algorithm that had to be added and changed to work with the Linux kernel. These have been tested to see that they work equally well in the kernel as outside. The tests were incorporated in the initialization of DCCP so that they would run on system start up. If there were any errors it would be printed with *printk* and could be checked with the command:

$ dmesg

Square root part

Java testing of the square root addition has proven that the addition works for all values of long. The result value is the same as the ceil value of *Math.sqrt* in Java, which is as good as it gets with the approach taken in section 3.2.1. Input and result values from the Java version where stored to be run with the kernel implementation of the square root algorithm. The result of the chosen values were calculated and they all returned equal results as their Java counterpart. This means the square root implementation works as expected.

Main part

Testing of the algorithm as a whole were also tested in the same way as the square root part. But in this case the input values can not be chosen as easy as for square root. Nevertheless, a substantial amount of variables were chosen and put through the calculation to do the comparison. As with the square root part, this test did not report any errors either.

5.1.2 Virtual testing

Virtual machines were set up with direct communication with one and other while implementing. The goal of these sort of tests are to see if the implementation works with simple sending and receiving.

A client and server script with python were used to send messages from one virtual machine to the other. If the message got through and were as it should on the server, the test were considered okay. The message sent with the compiled MulTFRC implementation is received as it were sent, meaning no errors with the communication on a protocol level.

Tests with Iperf were also made on these machines to see that they could withstand more data on each flow. This test was also passed. But the result can not be used because a virtual network can not be trusted.

5.1.3 Real environment testing

Three physical machines were set up to run tests within a real environment. Two of the machines will act as endpoint nodes. One of these will send data and the other will receive them. The third machine will be in the middle, between both these machines, acting as the router. It will introduce various different network scenarios to see how MulTFRC behaves. The scenario changes affects the delay, loss and bandwidth on the network. All of the machines have the same specifications that are listed in table below, with one exception. The router has two network interfaces instead of one.

The kernel has to be compiled on both endpoint nodes in order to test the implementation. The machine acting as a router can use the kernel version that comes with Debian as every data packet passing through it will never be sent to the transport layer. Both endpoint nodes has to make sure they have the same value of N before any data can be sent. Data between them is sent and received by the use of Iperf. Node 2 will act as a server

Table 5.1: Testbed hardware specifications

CPU	Intel Core 2 Duo E4500
CPU Speed	2.2 GHz
RAM	2 GB
Network	Intel® 82566DM Gigabit Ethernet

(receiver) and Node 1 as a client (sender). The network scenario also needs to be set before any sending of data can start. Which means the router needs to run a few commands to configure its network interference.

Configuration commands to set a network scenario on first network interface of the router:

$ tc qdisc del dev eth0 root

$ tc qdisc add dev eth0 handle 1: root tbf rate 10mbit burst 15000 limit 15000

$ tc qdisc add dev eth0 parent 1:1 handle 10: netem delay 50ms loss 0.25%

Configuration commands to set a network scenario on second network interface of the router:

$ tc qdisc del dev eth1 root

$ tc qdisc add dev eth1 handle 1: root tbf rate 10mbit burst 15000 limit 15000

$ tc qdisc add dev eth1 parent 1:1 handle 10: netem delay 50ms loss 0.25%

In this case a bandwidth limit of 10 Mbit is set. The total delay and loss needs to be summed up. By the endpoint nodes the delay is equal to 100 ms and the loss is equal to 0.5% in this case.

Commands needed to receive data on node 2:

$ sysctl -w net.dccp.default.multfrc_n=10000

$ iperf -d -s

With all router and server configurations set, the commands needed to start sending from node 1 are:

$ sysctl -w net.dccp.default.multfrc_n=10000

$ iperf -d -c 10.0.1.2 -t300 -l 1400B

When this is done, node 1 will now receive data with the DCCP protocol on the default port set by Iperf. 1400 bytes of data is sent in each data packet. This value is chosen because every packet below 1500 bytes has a guarantee that it will not be split up by any network entity. 1400 bytes combines with appended headers should keep it below this value and therefore avoiding potential splits. The value of N equal to 1 (scaled by 10000). The data stream is set to end after 5 minutes (300 seconds). All of these values, except N, are used as standard for all tests. An illustration of the flow can be viewed in figure 5.1.

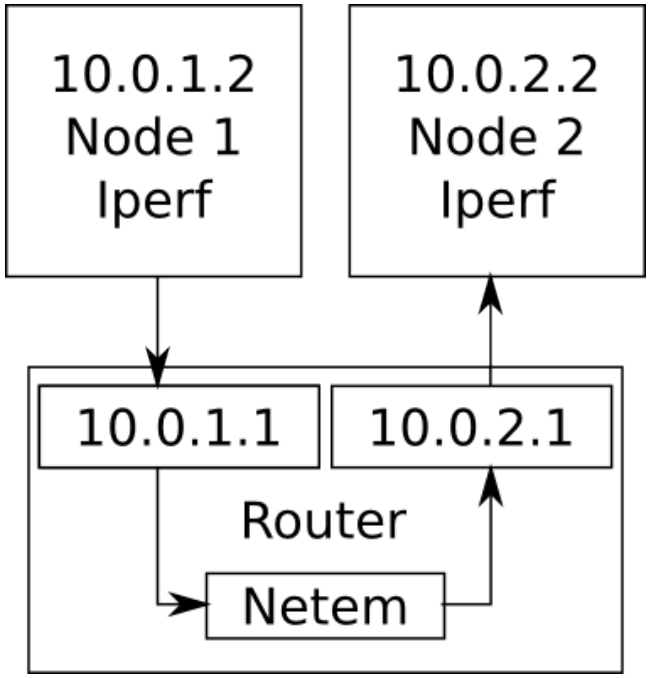


Figure 5.1: Testbed set up

Wifi network

There will be no testing of this implementation on a wireless network. Doing so can introduce unwanted error situations where a wired network would not. A wired network can guarantee that no other entities than those connected will interfere, where as a wireless network could potentially be disturbed by any close by unknown entity. As every aspect of the implementation can be fully tested with a wired network there is no reason to do the same with a wireless network where unknown factors can be introduced.

5.2 Evaluation

Tests to observe how the throughput of MulTFRC behaves in different loss scenarios uses Iperf to send data. Every test is ran with a bandwidth of

10 Mbit and delay of 100 ms. Most of the tests has been conducted with several flows at the same time. These tests are meant to compare TCP with MulTFRC or TFRC. *N* is used in the tests with MulTFRC while TFRC has increased number of flows to obtain the same desired goal. A few of the tests have been done with only one flow to observe how it performs whenever loss is introduced or changed.

All tests done with TCP uses the *cubic* congestion control algorithm. This is the standard for TCP in a Debian installation. The capture size of packets in Tcpdump had to be increased to run analysis on the collected data for TCP.

5.2.1 MulTFRC with N = 1 and 5% loss

In this test MulTFRC is ran alone to see how it behaves when the network loses 5% of the packets. The duration of this test is short in order to get a close look at the graph. The graph shows the amount of throughput in time. Figure 5.2 shows the result.

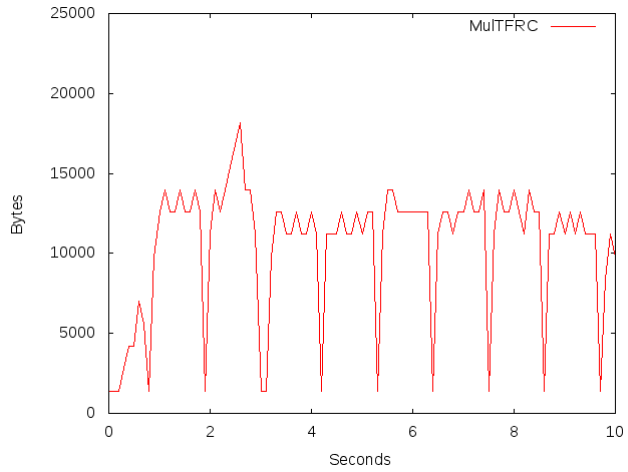


Figure 5.2: Throughput of one MulTFRC flow

The graph shows that the throughput is kept stable when it has found its pace. With the amount of periodic losses the flow seems to stabilize the sending rate at about 3 seconds. Whenever a loss occur after this point, the flow comes right back up to where it was before the loss. Thus keeping the flow of data at a stable rate and maintaining the smoothness that MulTFRC should have. At about 2.5 seconds the graph spikes a bit higher than what it does for the rest of this test. Likely there was not many, if any, lost packets at this point.

5.2.2 One TFRC flow and 5% loss

To compare the previous test of MulTFRC with TFRC the same test were applied. The result can be seen in figure 5.3.

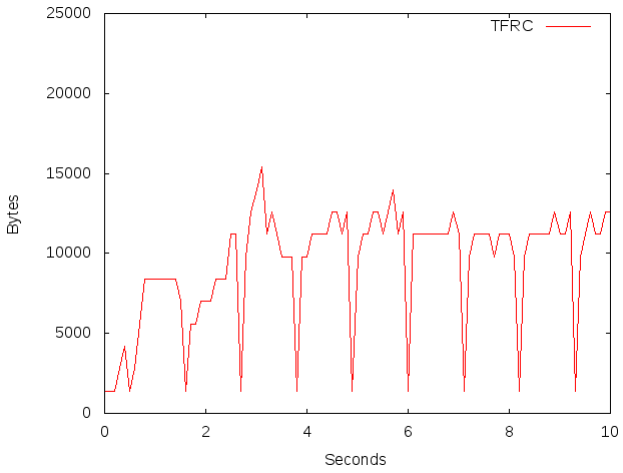


Figure 5.3: Throughput of one TFRC flow

The result closely resembles the result of MulTFRC. Compared to each other they both have near equal smoothness. Thus this test also shows that TFRC provides a stable sending rate for the flow. It also seems to have found its pace at close to the same time as MulTFRC. Looking closely one can see that this graph has two spikes. They occur at about 3 seconds and 5.5 seconds. If both of these tests were done with a longer duration one would probably see that the sending rate is very stable.

5.2.3 One TCP flow and 5% loss

To see how the two previous tests are different from TCP, the same test were also applied with TCP. The result can be seen in figure 5.4.

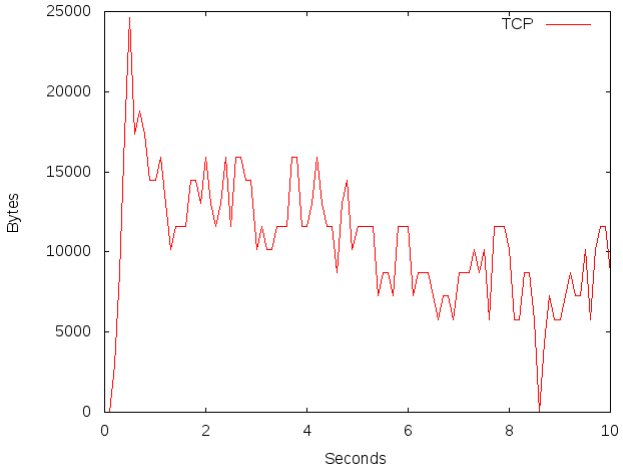


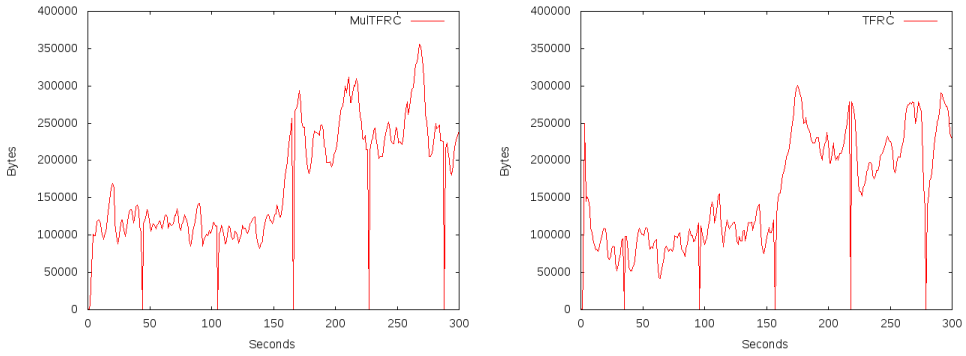
Figure 5.4: Throughput of one TCP flow

At first glance, the most obvious difference is the smoothness. There is much more variation of throughput in this graph and it also shows clearly

how TCP drops the sending rate after each loss. The throughput is also at times higher than any of the two other graphs. This should be due to the aggressiveness that TCP has. The result of this graph were as expected.

5.2.4 MulTFRC and TFRC with changing loss

The sending rate of MulTFRC should adapt to changes in the network in a similar manner to TFRC. The loss percent is changed from 5% to 1% halfway through this test to see how the sending rate of the flows change. The duration of the test is 5 minutes. The graph shows throughput in time. The result can be seen in figure 5.5.



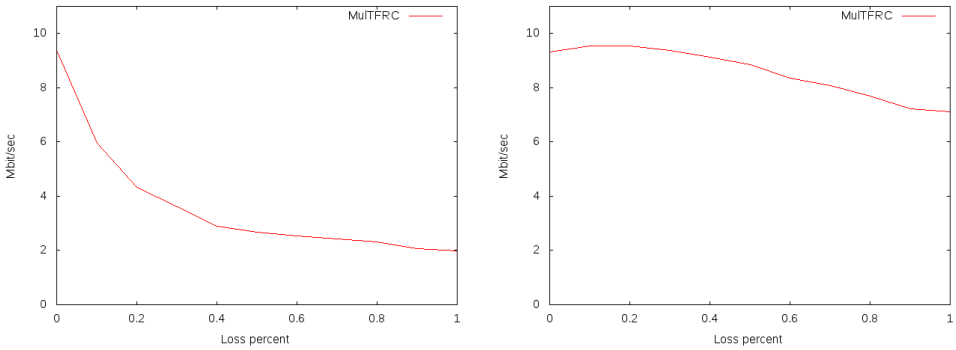
(a) MulTFRC (b) TFRC

Figure 5.5: Smoothness comparison of MulTFRC and TFRC with a change of loss from 5% to 1% at 150 seconds.

At the start of the graph in figure 5.5(b) it seems that TFRC builds up and that there are no to little loss to stop it from doing so. Thus the spike of high throughput at the start. This changes into something that resembles figure 5.3 more after the flow has found its pace. Aside from that, both graphs looks similar to figure 5.2 and 5.3. The spikes in throughput that occur after the loss has been changed to 1% seems to be within reason. It slowly builds up when there is no loss and falls down when loss happens. The important thing is that they both continues close to where it was before the loss, to keep the sending rate smooth. MulTFRC seems to adapt to the change in the network as it should and in a similar manner to TFRC. The sending rate also seems to be stable with little variation in both graphs.

5.2.5 Comparing MulTFRC N = 1 and N = 10

A flow with higher value of N is expected to deliver a data with a higher throughput if the network allows it. In figure 5.6 this is tested with one flow of MulTFRC in each sub figure. Figure 5.6(a) has been created with $N = 1$ and figure 5.6(b) with $N = 10$. The graph illustrates average throughput in time with a increasing loss percent. Every measured point of the graph has been monitored for 5 minutes to obtain an accurate average.



(a) MulTFRC with N = 1

(b) MulTFRC with N = 10

Figure 5.6: Comparison of MulTFRC with N = 1 and N = 10.

The flow with $N = 10$ is not ten times higher than the flow with $N = 1$. When the flow with $N = 1$ sends with a speed higher than and close to *2 Mbit/sec* at every point, it can not be expected that a flow can be ten times higher when the bandwidth is capped at *10 Mbit/sec*.

At the start of both graphs the measured throughput is close to each other. This was not expected and this occurrence will be discussed later. As the amount of loss is increased, the results is much more close to what one could expect. Figure 5.6(b) has a overall higher throughput than figure 5.6(a) when the loss is above *0.2%*.

5.2.6 MulTFRC N = 1 vs one TCP flow and loss up to 10%

This test compares the throughput of MulTFRC with $N = 1$ and one TCP flow. It starts out at 0% loss and increase by 1% each run. Every run lasts for 5 minutes. The average throughput per second is compared to the loss percent. The result can be observed in figure 5.7.

At 0% loss the TCP flow seems to be consuming most of the bandwidth. When loss is introduced the bandwidth seems to be shared more equally. Though MulTFRC is a bit higher overall after the loss introduction. Like in figure 5.6, the throughput at 0% loss was not expected.

5.2.7 MulTFRC N = 1 vs one TCP flow and loss below 1%

A closer look at the throughput below 1% loss is a natural consequence of the previous test. This test starts at 0% but increases by 0.1% for each run. The time for each test is still the same. The result can be observed in figure 5.8.

Like the test in figure 5.7 it seems to work fine with one TCP flow and MulTFRC with $N = 1$ (Figure 5.8(a)) after some loss is introduced. The same occurrence with *0%* loss can also be observed in this test as well. It seems to only be the case for runs very close to 0% as the throughput increases substantially from 0% to 0.1%. The fact that this also occurs with

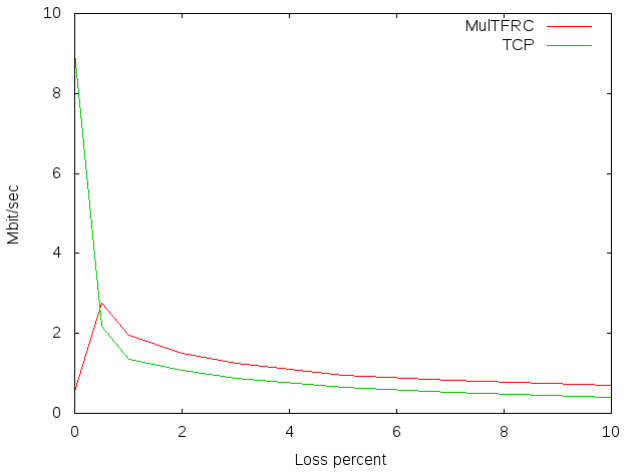
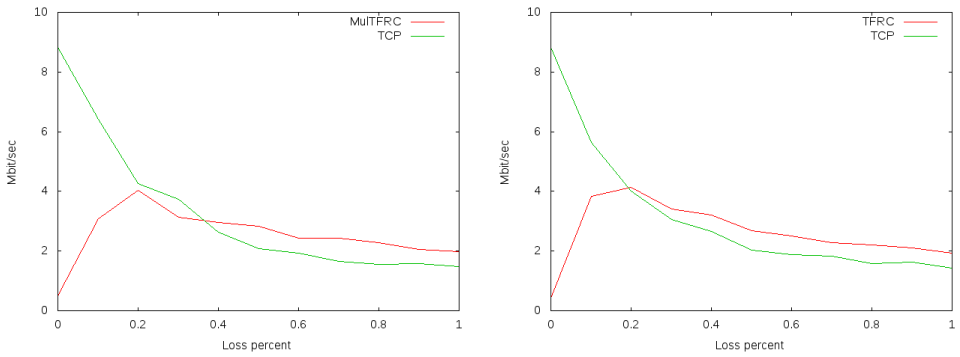


Figure 5.7: MulTFRC N = 1 versus one TCP flow



(a) MulTFRC N = 1 and one TCP flow

(b) One TFRC flow and one TCP flow

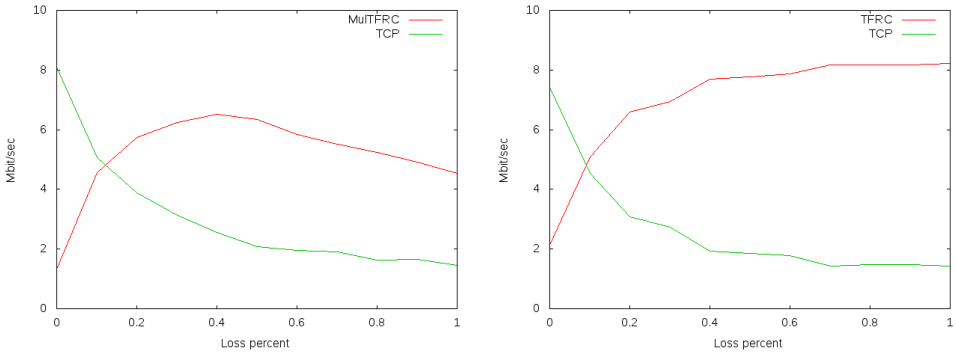
Figure 5.8: MulTFRC with N = 1 and one TFRC flow compared with one TCP flow separately.

TFRC in figure 5.8(b) means that this might not be caused by the MulTFRC implementation, but rather some flaw that already existed beforehand. This issue will be further discussed in a later section.

5.2.8 MulTFRC N = 5 and five TFRC flows compared to one TCP flow

MulTFRC with N set to 5 together with one TCP flow can be seen in figure 5.9(a), and five TFRC flows together with one TCP flow can be seen in figure 5.9(b). The duration of each test is set to 5 minutes. The graph shows average throughput with a given loss percent.

The same occurrence at 0% happens in this graph also. As this also happened with $N = 10$ in figure 5.6(b), it was not expected that changing the value of N would fix this. After some loss is introduced MulTFRC (Figure 5.9(a)) uses a sending rate that looks close to five time what TCP does.



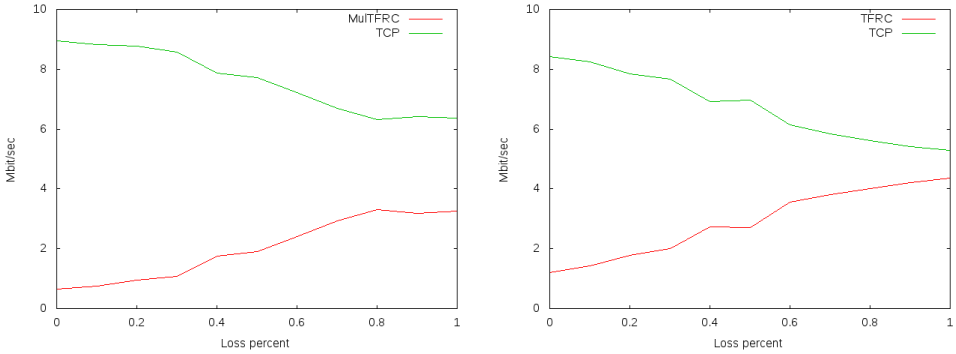
(a) MulTFRC N = 5 and one TCP flow (b) Five TFRC flows and one TCP flow

Figure 5.9: MulTFRC with N = 5 and five TFRC flows compared with one TCP flow separately.

In figure 5.9(b) TFRC increases while TCP decreases when the loss percent grows. In figure 5.9(a) MulTFRC and TCP both decrease in a similar manner. It is therefore shown that MulTFRC with $N = 5$ reflects the behavior of the TCP flow more than what five TFRC flows does.

5.2.9 MulTFRC N = 5 and five TFRC flows compared to five TCP flows

MulTFRC with N set to 5 together with five TCP flows can be seen in figure 5.10(a), and five TFRC flows together with five TCP flows can be seen in figure 5.10(b). The duration of each test is set to 5 minutes. The graph shows average throughput with increasing loss percent.

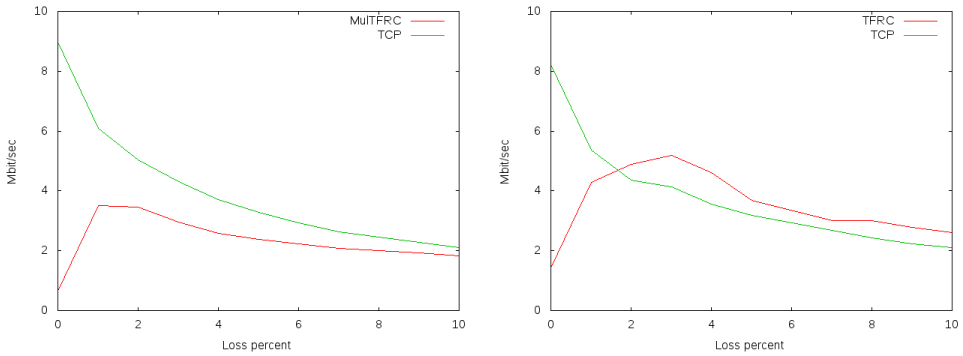


(a) MulTFRC N = 5 and five TCP flows (b) Five TFRC flows and five TCP flows

Figure 5.10: MulTFRC with N = 5 and five TFRC flows compared with five TCP flows separately.

Both MulTFRC and TFRC in figure 5.10 shows the same trends in their graphs. They both start out with a low throughput compared to TCP at

0% loss. The throughput increases for MulTFRC and TFRC and decreases for TCP while the loss is increased in both graphs. TFRC is closer than MulTFRC with TCP at *1%* loss. As a first, this graph looks to be of more interest after the *1%* loss mark. The same test is therefore reapplied with the same variables, except the loss percent. It was measured at every *1%* of loss up until *10%*. The result can be seen in figure 5.11.



(a) MulTFRC N = 5 and five TCP flows

(b) Five TFRC flows and five TCP flows

Figure 5.11: MulTFRC with N = 5 and five TFRC flows compared with five TCP flows separately. With higher values of loss.

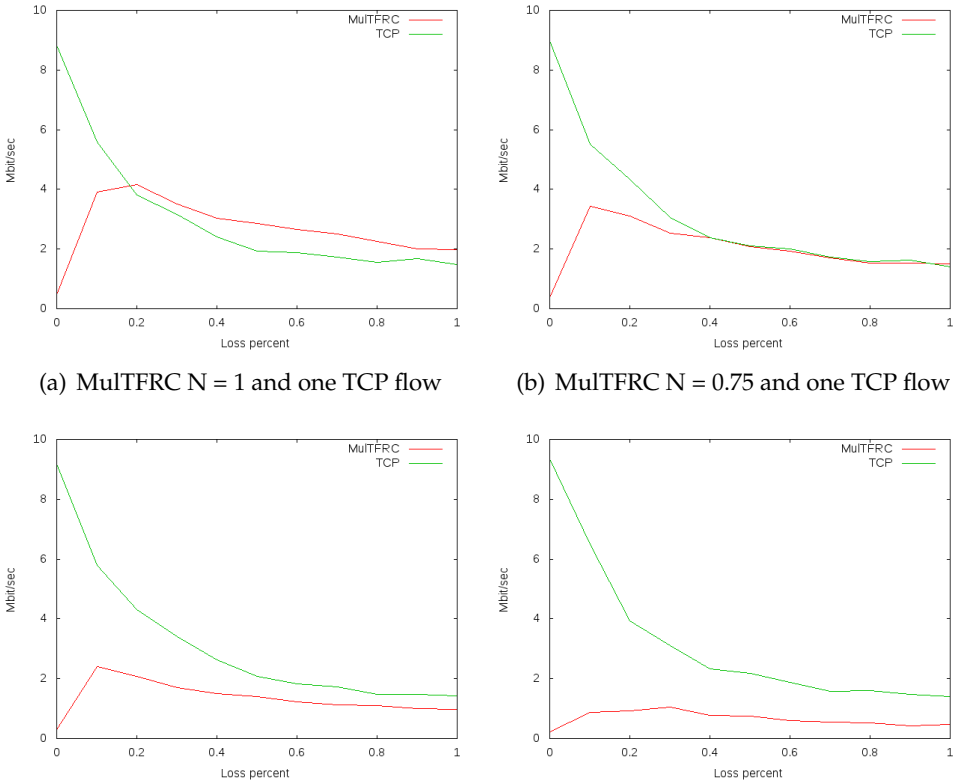
When the loss percent is increased to about *4%* the throughput is close to being equally shared in both figure 5.11(a) and figure 5.11(b). MulTFRC is just below TCP in throughput while TFRC is just above. The fact that both MulTFRC and TFRC showed the same early trend in figure 5.10 and figure 5.11 means that yet again this issue is not caused by the algorithm of MulTFRC, but rather something already existing in the Linux kernel implementation of DCCP.

5.2.10 MulTFRC with low N values together with one TCP flow

The figure 5.12 shows four graphs of MulTFRC with low values of *N* together with one TCP flow. All four shows average throughput per second with a given percent of loss. The loss percent is increased by *0.1%* between each test, starting at *0%* up to *1%*. Each measured point is the average throughput after 5 minutes of sending.

As also seen in previous measurements, all of the MulTFRC flows starts out with a low sending rate when the loss percent is close to *0%* while TCP does the opposite. The gap between the TCP flow and the MulTFRC flow closes when the loss percent increases. The measurements at *0.2%* percent loss and higher is near the expectations.

In figure 5.12(a) the MulTFRC flow catches up with the TCP flow at *0.2%*. After this point the MulTFRC flow has a higher, but close, throughput than the TCP flow. In figure 5.12(b) the MulTFRC flow and the TCP flow is close to equal from *0.4%* loss and higher. The decrease of throughput for the MulTFRC flow from figure 5.12(a) to figure 5.12(b) fits



(a) MulTFRC N = 1 and one TCP flow

(b) MulTFRC N = 0.75 and one TCP flow

(c) MulTFRC N = 0.5 and one TCP flow

(d) MulTFRC N = 0.25 and one TCP flow

Figure 5.12: MulTFRC with cumulative flows set to one and below compared with one TCP flow.

the values chosen for N. Although it was expected that a MulTFRC flow with $N = 1$ would be closer to one TCP flow than a MulTFRC flow with $N = 0.75$. In figure 5.12(c) and figure 5.12(d) the throughput continues to decrease accordingly and as expected.

The fact that the MulTFRC flows in figure 5.12 decreases with about one quarter in each sub figure proves that the values of N lower than or equal to 1 works as they should. It has been observed in figure 5.8(b) that one TFRC flow has a higher throughput than one TCP flow when the loss is greater than 0.2%. With this in mind, it is not wrong that MulTFRC with $N = 1$ is also a bit higher after the same loss percent. It is also positive that MulTFRC can be tweaked with the value of N to come even closer to the throughput of TCP than TFRC can.

5.2.11 Low throughput problem with MulTFRC and TFRC

In nearly all presented graphs both MulTFRC and TFRC has a low throughput combined with low loss percent and TCP. This issue seems not to be related to the algorithms implemented for MulTFRC and TFRC. Printing of input variables and calculated sending rate was turned on

while running some of the tests again to further investigate. It was then discovered that the input and result was not wrong, but that the algorithm was called less for runs with low loss percent. When the algorithms are hardly used it is hard to provide a sending rate based upon their calculation. This explains why both MulTFRC and TFRC uses so low sending rate compared to TCP at low loss percentages and behaves as expected when more loss is introduced. In figure 5.10 the sending rate of MulTFRC and TFRC does not increase as fast as when there is only one TCP flow. The reason for this is that five TCP flows combat congestion instead of one. This means the MulTFRC and TFRC flows will face congestion less often than they could with only one flow of TCP. Thus less calculations is done by the algorithms as they are not called as often.

Part III

Conclusion

Chapter 6

Conclusion

In the first part of this thesis MulTFRC was implemented in the Linux kernel. The original MulTFRC algorithm uses floating-point arithmetic to calculate the accurate sending rate. As the Linux kernel does not favour the use of floating-point arithmetic, the first problem was established. As a consequence the integer version had to be implemented. The needed addition of the square root to the algorithm was proven to work for all possible input values and does not affect the precision of the integer algorithm.

During testing of the integer version of the algorithm it was discovered that the percentage of when these two versions differed more than *10%* was *0.4%* of the time. Attempts were made to try and increase the precision of the integer algorithm without being successful. Further testing of the algorithm with reasonable and known input variables showed a slight decrease of occurrences. The maximum differential was at *54%* and might seem a bit high. But this happens very rarely and can therefore be accepted.

In the second part of the thesis the implementation of MulTFRC was tested and compared with TFRC. These tests shows most positive results, but also some negative. The negative results is not related to the implementation done in this thesis, but is an inherited flaw from the experimental DCCP implementation of the Linux kernel. The implementation of TFRC is untouched and shows the same problem related to tests run with a low percent of loss.

Tests shows that the expected smoothness is present for the MulTFRC implementation. Every time the loss is introduced on the connection MulTFRC adapts in a manner that is favoured for its intended applications, as it is suppose to. The increase and decrease of the sending rate as a result of loss is stable in all the tests that has been conducted.

Aside from the discovered flaw of DCCP, usage of the N variable to provide cumulative TFRC flows with MulTFRC works as intended. This has been proven for values of N less than, equal and greater than *1*. Based on the observations in this thesis it is concluded that the MulTFRC algorithm implementation works with the desired smoothness and cumulative flows.

Future directions

Further investigation towards solving the case were both TFRC and MulTFRC use a much lower sending rate than expected when there is a low percent of loss on the network is needed to make full use of MulTFRC. As already mentioned, it might be caused by the lack of calls to the congestion control equations.

Improvements to the integer version of the MulTFRC algorithm could help decrease the amount of times it differs from the floating-point algorithm. It would probably not change the result of any of the graphs obtained in this thesis because the occurrence is so low. But it would come of better if both floating-point and integer versions provides results that would differ less from each other.

An implementation of the floating-point version in the Linux kernel could also be a posed future direction. Both implementations could be put through a performance test where CPU time could be evaluated. It could be a nice addition to offer the floating-point version through the kernel configuration (*menuconfig*), giving the user more options. Even if the use of the FPU could be measured to calculate the sending rate slower, it would still be more precise then the currently implemented integer version. Thus it could be considered a valid, but not default, option.

During the testing part of this thesis tools for DCCP where in short stock. Developing tools for testing DCCP in network environments would prove valuable if the usage of DCCP increases. Iperf and Tcpdump were the two only useful tools found during the course of this thesis. Tcpdump needed no patching to capture packets, but lacked tools to analyse the monitored results.

Bibliography

[1] Dragana Damjanovic. *Parallel TCP Data Transfers: A Practical Model and its Application*. Feb. 2010. URL: http://home.ifi.uio.no/michawe/research/projects/multfrc/dragana_thesis.pdf.

[2] Michael Welzl Dragana Damjanovic. *MulTFRC: Providing Weighted Fairness for Multimedia Applications (and others too!)* July 2009. URL: http://heim.ifi.uio.no/michawe/research/publications/ccr-multfrc.pdf.

[3] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard). Updated by RFCs 2817, 5785, 6266, 6585. Internet Engineering Task Force, June 1999. URL: http://www.ietf.org/rfc/rfc2616.txt.

[4] S. Floyd, E. Kohler and J. Padhye. *Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC)*. RFC 4342 (Proposed Standard). Updated by RFCs 5348, 6323. Internet Engineering Task Force, Mar. 2006. URL: http://www.ietf.org/rfc/rfc4342.txt.

[5] S. Floyd et al. *TCP Friendly Rate Control (TFRC): Protocol Specification*. RFC 5348 (Proposed Standard). Internet Engineering Task Force, Sept. 2008. URL: http://www.ietf.org/rfc/rfc5348.txt.

[6] M. Handley et al. *TCP Friendly Rate Control (TFRC): Protocol Specification*. RFC 3448 (Proposed Standard). Obsoleted by RFC 5348. Internet Engineering Task Force, Jan. 2003. URL: http://www.ietf.org/rfc/rfc3448.txt.

[7] E. Kohler, M. Handley and S. Floyd. *Datagram Congestion Control Protocol (DCCP)*. RFC 4340 (Proposed Standard). Updated by RFCs 5595, 5596, 6335, 6773. Internet Engineering Task Force, Mar. 2006. URL: http://www.ietf.org/rfc/rfc4340.txt.

[8] Stein Gjessing Michael Welzl Dragana Damjanovic. *MulTFRC: TFRC with weighted fairness*. Internet Engineering Task Force, July 2010. URL: http://tools.ietf.org/html/draft-irtf-iccrg-multfrc-01.

[9] P.V. Mockapetris. *Domain names - implementation and specification*. RFC 1035 (INTERNET STANDARD). Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604. Internet Engineering Task Force, Nov. 1987. URL: http://www.ietf.org/rfc/rfc1035.txt.

[10] J. Postel. *Transmission Control Protocol*. RFC 793 (INTERNET STANDARD). Updated by RFCs 1122, 3168, 6093, 6528. Internet Engineering Task Force, Sept. 1981. URL: http://www.ietf.org/rfc/rfc793.txt.

[11] J. Postel. *User Datagram Protocol*. RFC 768 (INTERNET STANDARD). Internet Engineering Task Force, Aug. 1980. URL: http://www.ietf.org/rfc/rfc768.txt.

[12] K. Ramakrishnan, S. Floyd and D. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168 (Proposed Standard). Updated by RFCs 4301, 6040. Internet Engineering Task Force, Sept. 2001. URL: http://www.ietf.org/rfc/rfc3168.txt.